Online Anomaly Detection for Reconfigurable Self-X Real-Time Operating Systems

A Danger Theory-Inspired Approach

Dissertation

A thesis submitted to the
Faculty of Electrical Engineering, Computer Science and Mathematics
of the
University of Paderborn
in partial fulfillment of the requirements for the degree of
doctorum rerum naturalium (Dr. rer. nat.)

Diplom-Informatikerin Katharina Stahl

November 14, 2015

Acknowledgements

Es ist fertig, endlich! Und genau wie jeder Andere, der eine Dissertation verfasst hat, habe ich auch alle Phasen durchlebt: von Enthusiasmus für das Thema, den Zweifel eine Lösung zu finden oder diese umsetzen zu können, Gedanken übers Aufgeben - es einfach sein zu lassen, der Schwierigkeit den inneren Schweinehund zu überwinden und an der Sache dran zu bleiben, bis schließlich hin zum Einfach-nur-Durchzuhalten, um die Sache abzuschließen. Auf diesem Weg haben mich viele Menschen aus meinem Leben begleitet und unterstützt - jeder in seiner eigenen Rolle -, und denen möchte ich auf diesem Wege danken:

Ein ganz großer Dank geht als erstes an meinen Doktorvater Prof. Franz Rammig, der mir nicht nur die Möglichkeit zur Promotion gegeben hat, sondern mich wirklich auf dem Weg dahin begleitet hat. Wahrscheinlich hatte er zwischenzeitig genauso große Zweifel wie ich, ob ich denn irgendwann zu einer passenden Lösung komme. Nichtsdestotrotz hat er mich immer wieder ermuntert und mir gute Hinweise gegeben und viele Gespräche über Ideen und den Fortschritt der Arbeit mit mir geführt - eben genau so, wie man sich einen Doktorvater vorstellt! Auch dafür, dass er in seiner Rolle als Gutachter die doch so umfangreich gewordene Arbeit gelesen hat - und manche Kapitel auch mehrfach: Herzlichen Dank!

Zur Umsetzung dieser Arbeit haben auch weitere Personen beigetragen: angefangen von Gavin Vaz und Sijia Li, die durch ihre Master-Arbeiten die Vorarbeiten zur Implementierung meines Konzepts geliefert haben, und insbesondere Sijia Li, der als studentische Hilfskraft große Teile der Programmierung übernommen hat. Insbesondere aber ohne die großartige Unterstützung von Jörg Stöcklein wäre eine Evaluierung meiner Arbeit nicht möglich gewesen. Jörg hat sich nicht nur die Ideen zur Umsetzung angehört und mit mir diskutiert, vielmehr hat er mir die Evaluierungsumgebung zur Verfügung gestellt und sie an die Bedürfnisse meiner Case Study angepasst, ohne dass es für ihn einen persönlichen Nutzen hatte. Das war wirklich großartig!

Meiner lieben Schwägerin, Silke Stahl-Rolf, muss ich herzlich dafür danken, dass sie die komplette Arbeit durchgearbeteitet hat und mich darin unterstützt hat, Rechtschreib- und Tippfehler zu entfernen oder zumindest ihre Anzahl zu reduzieren.

Ich danke meinen Eltern nicht nur dafür, dass sie den Weg bereitet haben, bis hier hin zu kommen. Insbesondere danke ich Ihnen dafür, dass sie mir immer noch tagtäglich ihre Unterstützung anbieten, sich um unsere Kinder kümmern, und mir dadurch viel Freiraum geschaffen haben, mir Zeit für die Fertigstellung dieser Arbeit zu nehmen.

Auch mein Mann Jan musste insbesondere im letzten Jahr viel Einsatz zeigen, im Bezug auf die Kinder sowie den Haushalt - was er beides großartig gemanaged hat, während ich mich ins Arbeitszimmer verzog. Was aber viel wichtiger für mich war, war sein unerschütterlicher Glaube an mich und an die Tatsache, dass ich es schaffen würde, - auch wenn ich selbst zwischendurch große Zweifel hatte.

Ein ganz besonderer dicker Kuss geht an meine beiden kleinen Töchter Amalia (6) und Paulina (5), die vielleicht noch nicht verstehen, dass Mama "Doktor" werden möchte aber NICHT Arzt, - die füer ihr Alter viel Verständnis und Geduld aufweisen mussten, wenn ich statt spielen schreiben musste und mir aber dennoch zur Unterstützung diese zwei tollen Bilder ausgemalt haben:

Abstract

Anomaly detection is considered as a useful means to enhance the system's run-time dependability in self-reconfiguring real-time systems. Employed in self-reconfiguring real-time systems, anomaly detection requires to fulfill the specific requirements and challenges resulting from the specific characteristics of the application domain: working online, being lightweight in terms of resource consumption, self-learning, and, in order to be able to cope with dynamically changing behavior, it requires a context-related evaluation of behavior.

These requirements have been addressed by the Online Anomaly Detection as the central contribution of this thesis. Its concept was inspired by the Danger Theory coming from Artificial Immune Systems and building up an operating system framework for the context-related classification of behavior. Composed of system call sequences executed by the applications tasks, the behavior is evaluated on the basis of input signals reflecting the operating system state determined as the behavior's context. A compact data structure on the basis of Suffix Trees has been integrated into the framework for building up a Behavior Knowledge Base to store the behavior history. Based on the properties of Suffix Trees, it enables online profiling of the behavior including its immediate conservation.

The applicability of the Online Anomaly Detection with respect to the given restrictions was proven by formal analysis of the approach and it was verified by a quantitive evaluation of the implementation of the approach within the real-time operating system ORCOS. Furthermore, the performance of the Online Anomaly Detection in terms of its effectiveness was demonstrated by means of a case study using the autonomous BeBot application within a virtual evaluation environment.

The evaluations have shown that the Online Anomaly Detection offers a powerful concept to enhance the run-time dependability of self-reconfiguring real-time systems by implementing context-related classification of system behavior and, beyond that, by providing means to identify potential threat sources in conjunction with the evaluation of the system health state.

Zusammenfassung

Anomaliedetektion ist ein wertvolles Mittel zur Verbesserung der Laufzeitzuverlässigkeit in selbst-rekonfigurierbaren Echtzeitsystemen. Aufgrund der spezifischen Eigenschaften, die sich aus der Kombination von selbst-rekonfigurierbared Systemen und Echtzeitsystemen ergeben, muss die eingesetzte Anomaliedetektion spezifische Anforderungen erfüllen: online-fähig sein, leichtgewichtig im Sinne des Verbrauchs von Ressourcen und selbst-lernend. Um mit sich dynamisch ändernden Verhalten umgehen zu können, muss die Anomaliedetektion eine kontextbezogene Evaluierung des Systemverhaltens umsetzen.

Der zentrale Kern dieser Arbeit ist die Online Anomaly Detection, die auf Basis dieser spezifischen Anforderungen entworfen wurde. Das Konzept der Online Anomaly Detection beruht auf der Danger Theory, welche zu den Verfahren der Künstlichen Immunsysteme gehört. Das Konzept stellt ein Betriebssystem-Framework zur kontext-bezogenen Klassifizierung von Systemverhalten zur Verfügung. Dabei setzt sich das Systemverhalten aus den Systemaufrufen der Anwendungen zusammen und wird auf Basis von Eingangssignalen evaluiert, welche den Betriebssystemzustand als Kontext des Systemverhaltens widerspiegeln. Zur Speicherung der Historie des Systemverhaltens wurde in das Framework eine Wissensbasis (Behavior Knowledge Base) integriert, die mithilfe von Suffixbäumen eine kompakte Datenstruktur erzeugt. Die Verwendung von Suffixbäumen ermöglicht einen Abgleich des Systemverhaltens zur Laufzeit und dessen direkte Ablage in die Wissensbasis.

Die Anwendbarkeit der Online Anomaly Detection auf das gegebene Einsatzgebiet wurde im Bezug auf die vorgegebenen Restriktionen anhand einer formalen Analyse bewiesen. Darüber hinaus wurde die Online Anomaly Detection im Echtzeitbetriebssystem ORCOS umgesetzt und zur Verifizierung der Anwendbarkeit eine quantitativen Evaluierung durchgeführt. Die Performanz und Effektivität der Online Anomaly Detection wurde im Rahmen einer Fallstudie demonstriert, welche auf Basis einer autonomen BeBot Anwendung innerhalb einer Virtuellen Evaluierungsumgebung durchgeführt wurde.

The Evaluierung des Ansatzes hat gezeigt, dass die Online Anomaly Detection ein erfolgreiches und leistungsstarkes Konzept bietet, um Laufzeitzuverlässigkeit in selbst-rekonfigurierenden Echtzeitbetriebssystemen zu erhöhen. Es ermöglicht kontext-bezogene Klassifizierung von Systemverhalten. Darüber hinaus bietet es Möglichkeiten zur Identifikation der Fehlerquellen, die auf Basis der Evaluierung des Betriebssystemzustandes erfolgt.

Contents

Part I

Introduction

CHAPTER 1

Introduction

Motivation

Recent developments enforce embedded systems - that predominantly are real-time systems - to become more and more intelligent and to implement capabilities to be able to cope with dynamically changing environments as well as dynamically changing requirements. As a means to address these challenges, embedded systems implement autonomous behavior by so called self-x mechanisms, like self-organization, self-optimization, self-healing etc. Regarded to be promising approaches, they offer potentials to the system to handle the challenge of changing operating conditions and to initiate according *intelligent* adaptations that change the system behavior without any external control. Autonomous execution decisions are also governed by factors like interactions between the system components and/or the applications. Self-x capabilities introduce high complexity in the behavior of the system. Because of this complexity, autonomous behavior can cause decisions that lead to previously unknown behavior and/or to unspecified system states for which it is unknown whether these novel behaviors or system states are stable. Hence, self-x behavior also introduces novel risks in terms of dependability. Dependability in embedded systems, however, is a crucial issue and must be ensured during the entire system life cycle.

Numerous investigations concentrate on the development of methods for autonomous systems to ensure the dependability at specification or design time since design time faults are permanent and will remain in the system. However, in autonomous systems, the actual performance becomes apparent when the system is put into execution. The behavior of an autonomous system is unpredictable and offers potentials to generate decisions that could also manifest suspicious or even malicious behavior during run-time. Hence, for autonomous systems it is essential that the system behavior, the autonomous decisions and the adaptations are evaluated and verified at runtime in order to ensure that at any point of time of the execution the system behavior is matching its intended specifications.

To address this, powerful methods like Hazard Analysis [80] and Online Model Checking [95] support autonomous reconfigurations and aim to verify and control the correct execution of the software system. These methods basically rely on specification knowledge or on offline generated system models. The system state analysis performed by Hazard Analysis is computed on an exhaustive system model and, therefore, is resource intensive. For this reason, it can only be executed offline or outside the system if some online operating is intended to be achieved. Online Model Checking [95] is an approach integrated as an operating system service for ensuring dependability at run-time inside the system itself. This method uses the actual system execution state which is established by system state information examined at specific time stamps to check the correctness of the execution trace by means of a partial system model. Choosing the time stamps for verification is a challenging issue in order to ensure an adequate granularity and not to miss any significant execution traces. Moreover, as both approaches are based on system models, they might be prone to be imprecise as models are used to describe abstractions of the real code. Applying models that may be imprecise could produce gaps between the system model and the real system behavior. On the other hand, we assume that the behavior of such complex systems being based on interaction, mobility, communication, environmental changes, and self-reconfiguring software (but also hardware) can neither easily be completely modelled at design time by a system designer nor by an automated development tool. Because of the system dynamics, autonomous decisions may lead to system states or behaviors that could have not been foreseen, and would therefore be not included when building up the model. Hence, there is always a risk that such approaches operate on models which might be incomplete so that the system may exhibit behaviors that are undefined. The questions remains how to deal with undefined states or behaviors as they do not need to be categorically incorrect or malicious in all cases.

The main motivation of this thesis is to overcome the limitations of specification-based approaches, and enable analyzing and evaluating autonomous system behavior decoupled from any previously specified model in order to close the gap and thereby enhance the system's dependability.

Objective

This thesis addresses the problem of run-time dependability of self-x real-time systems. As a consequence of avoiding the usage of design-time system models, the objective is to achieve run-time dependability based on real execution data. System run-time behavior has to be observed, analyzed and classified in order to evaluate behavior or system states and detect instablilities caused by the system dynamics or by autonomous system and application behavior. These properties and characteristics can be found as a principle of Anomaly Detection which motivated this thesis to investigate into this topic.

Run-time information, (state) parameters and execution data of a system and its applications exploited by anomaly detection are usually kept by the operating system that builds up the interface between the hardware platform and the executing application software and is responsible for their correct operation. In the context of real-time systems, real-time operating systems are applied to manage the execution of real-time applications as they provide mechanisms to ensure the timeliness and determinism of the real-time system. Consequently, the real-time operating system keeps the according run-time execution data of the real-time system appropriate to supply an Anomaly Detection module with input data. Numerous substantial works in the domain of Anomaly Detection exist. However, only a small subset is addressing real-time systems. Moreover, self-x systems are setting different requirements on the characteristics of an Anomaly Detection as behavior is intended to change dynamically.

The specifics of self-x environments set another challenges to real-time operating systems. In order to provide an adequate execution platform able to cope with dynamically changing conditions and applications that implement self-x capabilities, real-time operating systems must implement self-x capabilities by themselves [Stahl et al., 2014b]. They become self-x real-time operating systems. Self-x systems operate based on data monitoring of the system execution. Hence, the run-time execution data of a self-x real-time system may be extended by further dimensions effecting the input data of the Anomaly Detection. In particular, for self-x real-time systems, no Anomaly Detection approach could be identified to meet the specific requirements and challenges (described in the next section) to be integrated into a self-x real-time operating system. Based on this, with this thesis I am addressing that open issue and propose an approach for Anomaly Detection for self-x real-time operating systems.

1.1 Requirements, Restrictions and Challenges

The purpose of Anomaly Detection is to distinguish between harmless and harmful system activity, or to formulate this in the terminology of Anomaly Detection, it aims to differentiate between *normal* and *anomaly*. The application domain of this thesis are self-x real-time operating systems that have specific characteristics resulting in specific requirements and challenges on Anomaly Detection intended for those systems. In particular, in this thesis we combine two domains, namely self-x computing and real-time computing, that both entail different requirements which all have to be incorporated in the concept of an Anomaly Detection:

1. **Requirements from real-time systems**

   Kopetz defines a real-time computer system as ”a computer system where the correctness of the system behavior depends not only on the logical results of the computations, but also on the physical time when these results are produced” [63]. Hence, computer systems with real-time capabilities are bounded on deterministic timely behavior. The real-time operating system is responsible to manage the execution of application tasks with respect to their timing constraints called *deadlines*. For this purpose, it has a Scheduler that implements an advanced algorithm able to compute a schedule based on the timing parameters and priorities of the application tasks. In addition, a real-time operating system requires all its operating system components to be deterministic in their timely behavior as well.

   Embedded real-time systems are usually applied in critical environments executing mission-critical or even safety-critical applications. Hence, dependability is an important and fundamental system property, as a system failure may lead to severe consequences. The targeted Anomaly Detection is considered as a means to complement the entire set of dependability measures. Because of the criticality of the application domain of embedded real-time systems, a failure or a system behavior that might lead into a failure has to be detected as soon as it occurs, preferably before it is able to propagate throughout the system and probably effect the entire system’s operation reliability. Addressing this, one major requirement is that the Anomaly Detection is able to process the collection and evaluation of the execution data on demand working in an online manner.

   Embedded systems are usually resource-restricted systems, predominantly concerning memory space and computation time. An Anomaly Detection being integrated into a real-time system has to cope with these restrictions. Because of restricted memory space, it should not store huge data bases. Concerning computation time, the analyzing algorithms applied have to maintain the determinism of the system. In fact, both aspects interact mutually: huge data amounts to be analyzed by an Anomaly Detection usually effect the computation time of the analysis.

Obviously, any approach that operates online impacts on the execution times of the entire system including all effected components. The Anomaly Detection intended to work online introduces additional computation load that affects the execution of the system and, in particular, of the deadline-bounded application tasks. In order to avoid an unpredictable increase in computation times of application tasks that could lead into deadline misses, the computation time of the Anomaly Detection must be bounded.

All in all, an Anomaly Detection approach for real-time systems is required to work **online** and has to fulfill requirements of **bounded memory usage** and **bounded computation time**.

A further specific characteristic of tasks designed for and executed on a real-time operating system is that they are reactive, and in most cases even **periodic**. A task is defined by its period and its worst-case execution time and is executed in forms of task's instances assigned to each period. The periodic execution of tasks differs from the execution of processes for conventional computer systems. This fact mainly contributes to the representation of application behavior to be evaluated by Anomaly Detection.

2. **Requirements from self-x systems**

The term self-x encompasses properties like self-organization, self-optimization, self-adaptation, self-reconfiguration, etc. that are described in [24]. Self-x system are complex and face many internal and external effects and influences. Computer systems equipped with self-x capabilities are often classified as autonomous systems distinguished by their decision making process: the decisions made by the system are not human-controlled but are performed by the system itself in an autonomous manner. Self-x systems operate in an autonomous manner by applying a control loop consisting of sensing and monitoring, analyzing, and controlling. The decision process relies on data obtained by sensing of the system itself and its environment, and is based on predefined rules that may also be regulated by random factors, in order to meet the system's specified objective function. Often, autonomous systems are applied to complex problem solving for which conventional methods are not sufficiently applicable.

Operating systems designed for self-x applications must provide self-x capabilities by itself in order to react on dynamical changes of requirements and behavior and ensure quality of service. Therefore, Anomaly Detection integrated in such an operating system has also to work in an **autonomous** manner. Moreover, this requirement is reinforced by the requirement on Anomaly Detection to work online without engaging a human operator in order to prevent a delay of the system operation.

Implementing self-x capabilities leads to dynamical changes of system behaviors and/or configurations, also such ones that could not have been predicted in advance as they were produced based on autonomous decisions. Novel behaviors and system configurations are not autonomically potential system threats, - as it is usually assumed by many classical anomaly detection systems. However, not all resulting behaviors or configurations can be considered as stable or safe. Because of false specifications, autonomous decisions could also lead to system behaviors or configurations that can harm the dependability of the system. Further on, a reconfiguration of a particular system component caused by changing system requirements may unintendedly disequilibrate applications whose behavior was unaffected by the reconfiguration. Hence, autonomous changes in system behavior and the system configuration raise the most essential challenge on the Anomaly Detection. The Anomaly Detection must implement **self-learning** capabilities in order to be able to adjust the classification of known behaviors. Concurrently, it must be **able to cope with novel behaviors and system states**, examine their effect on the system and classify them.

An Anomaly Detection designed for self-x real-time operating systems must mandatorily meet all requirements from the two domains illustrated above. In summary, Anomaly Detection must be:

- online
- autonomous and, in particular, self-learning
- lightweight, meaning
  - bounded in terms of memory usage
  - bounded in terms of computation complexity and restricted in terms of computation time
- able to take into account the specific characteristics of real-time tasks of being periodic
- able to cope with unknown system behavior and system states and able to classify them correctly.

The basic motivation of this thesis is to overcome the problems concerned with specification-time models. Hence, this approach is intended to rely only on the execution time data without any (previous) knowledge about the application behavior. In order to be able to integrate Anomaly Detection into systems with already existing applications without the need of recompiling, but also to prevent a change in the development process of future applications in terms of interfaces, etc., an essential requirement is to realize the desired Anomaly Detection without any application-specific knowledge. Hence, Anomaly Detection is required to operate only such data that is available in the operating system.

1.2 Basic Idea

The challenge in developing an Anomaly Detection for self-x real-time operating systems is twofold: On the one hand, it requires a definition of behavior as the key point of interest of the Anomaly Detection. Related to this, an adequate representation of the behavior data must be determined in combination with a data structure for its storage. On the other hand, the core entity of the Anomaly Detection is the method applied for analysis and classification. For being applicable in the domain of self-x systems, the classification method is not allowed to only distinguish between known and unknown behavior. Moreover, it has to classify each behavior with respect to its potentials to harm the system.

The Anomaly Detection approach proposed in this thesis was inspired by the *Danger Theory* from *Artificial Immune Systems* which are known as a powerful but lightweight defense mechanisms with autonomous and self-learning capabilities. The Danger Theory, in particular, provides mechanism to tolerate (also novel) behavior that is classified as harmless while in contrary it induces a system reaction in case of harmful effects and thereby allows a context-related classification of behavior.

In this approach, behavior is represented by system call sequences executed by the applications as system calls predominantly affect state changes in the operating system. The threat potential of a behavior is reflected by the state parameters of the operating system associated with a system call execution. For storing the behavioral sequences, we use *Suffix Trees* as they enable a compact storage of recurring patterns that are identified by a low-cost pattern matching mechanism. The concept of the Suffix Tree is extended to hold the classification-related data associated with observed behaviors.

This basic idea is worked out in details in this thesis. It was implemented and integrated into a real-time operating system, and evaluated by means of a case study that show promising results.

Structure of this Thesis

This thesis is organized in parts enclosing the contentual contributions:

1. Part I describes the motivation of this thesis, determines the requirements and introduces the basic idea for the approach in Chapter 1.
2. In Part II, the foundations of this thesis are presented. A detailed introduction into Anomaly Detection is given in Chapter 2, describing properties, models, techniques and applications domains. Then, the basic of Artificial Immune System are discussed in Chapter 3 including an introduction into the Danger Theory. Afterwards, an overview over existing approaches is given in the Chapter 4, including concepts and approaches that inspired this thesis. Chapter 5 deals with Online Pattern Matching methods that are required for the behavior pattern matching used by the approach presented in this thesis. Finally, we introduce the real-time operating system ORCOS in Chapter 6 as it forms the basis of the implementation of the approach presented here.
3. Part III gives a detailed discussion of the concept and the motivation for using the selected techniques.
4. In Part IV, the implementation of the approach integrated into the real-time operating system ORCOS is presented in detail in Chapter 8, followed by a short evaluation of the costs in terms of memory and computation time in Chapter 9.
5. Part V presents the evaluation of the performance of the approach in forms of a case study. Chapter 10 gives a short introduction into evaluation methods of autonomous approaches and provides a motivation for using virtual environments as the one which we applied and described in Chapter 11. The results of the evaluation of the Anomaly Detection are presented in Chapter 12.
6. We conclude this thesis in Part VI with a summary and discussion of its contribution.

Part II

Foundations

CHAPTER 2

Anomaly Detection

In this chapter, we introduce anomaly detection, the existing techniques and their application purposes in order to be able to classify the anomaly detection approach proposed in this thesis. Even though anomaly detection, in general, has gained great attention in research in the recent years, there is no standard literature on anomaly detection. This is mainly caused by the fact that the problem of anomaly detection has often been related to its specific application domain. In [34], Chandola et al. give an overview in form of a technical report on anomaly detection and the existing techniques. The content of this chapter mainly refers to that report [34], but it also includes content provided by Patcha and Park [79], the book by Tan et al. [87] and a survey by Axelsson [22].

2.1 Definition

Many definitions or descriptions of Anomaly Detection exist in the literature, all of them being quite similar. In [87], Tan et al. give a description for Anomaly Detection being "the task of identifying observations whose characteristics are significantly different from the rest of the data." According to Chandola et al. in [34], "Anomaly Detection refers to the problem of finding patterns in data that do not conform to expected behavior." Another, more precise description of anomaly detection which is used by Chandola et al. (in [34]) and also by others like Patcha and Park (see [79]), is the ability of detecting deviations from a defined normal behavior or a so called *normal profile*.

Referring to these definitions and descriptions, the aim of anomaly detection is to recognize and identify significant information indicating changes in data or behavior during system operation. Thereby, any observation that does not match the *normal profile* is defined as an *anomaly*. By delivering this information, anomaly detection offers potentials to draw conclusion on qualities of system states at run-time.

Related terms in different application domains for anomalies are outliers, novelties, exceptions, aberrations, surprises, peculiarities, contaminants, noise, etc. so that the problem of anomaly detection is strongly related to research domains such as *novelty detection* and *outlier detection* [73, 74].

2.2 Properties and Features

Referring to the definition provided by the section above, anomaly detection systems rely on

1. normal profiles,
2. observation data and
3. the ability to detect deviations in the observation data.

This definition gives a general understanding of anomaly detection but it leaves it open to offer a precise specification. This is due to the fact that miscellaneous requirements originating from application purpose and problem characteristics are set on anomaly detection systems.

This section aims to classify anomaly detection problems with respect to requirements and problem characteristics that arise in various application domains. The problem classification is mapped on properties and features implemented by anomaly detection techniques. This in turn deals as a basis, on the one hand, to asses the effectiveness of anomaly detection techniques to specific problems, and, on the other hand, as a framework for the development of novel anomaly detection approaches.

2.2.1 Aspects of Problem Formulation

As a basis, Chandola et al. (in [34]) specify four main aspects to be determined concerning the anomaly detection problem and its application purpose: Nature of Data, Labels, Anomaly Type and Output that are explained as following:

1. **Nature of Data**

   The question of the nature of data is related to the data that can be obtained from the application and deals as input for anomaly detection. Tan et al. classify data and data types in detail in [87, Chapter 2] in the context of data mining, but as data collection is a prerequisite for anomaly detection, the classification made by Tan et al. can be adopted to the problem of anomaly detection. Data items are mainly differentiated into qualitative data that is composed of **symbolic** or **categorical** attributes and quantitative data that is represented by **numerical** attributes. The type of attributes thereby determines the properties of the attributes and their values. Tan et al. define four type of attributes:

   - **nominal**: nominal attributes provide information to distinguish different objects from one another, such as name or ID. Main operations on such attributes are $=$ or $\neq$.
   - **ordinal**: ordinal attributes extend nominal attributes as they contain enough information for ordering objects. Thus, operations allowed on such attributes are also $<$ and $>$.
   - **interval**: interval attributes allow to identify meaningful differences between objects with operations like $+$ and $-$; one example for interval attributes are calendar dates.
   - **ratio**: for ratio attributes, both differences and ratios are meaningful (with the operations $\cdot$ and $/$). These attributes allow to calculate means or variations, for example.

Nominal and ordinal attributes belong to **symbolic** or **categorical** attributes while interval and ratio attributes are **numerical** attributes.

A further distinction introduced by Tan et al. is the number of values the attributes can take. They differentiate between:

- **discrete**: discrete attributes can take either a finite set of values or a countably infinite set of values. They can be of categorical or numeric type.
- **binary**: binary attributes are a special type of discrete attributes as they can take only two different values ($true$ or $false$, 0 or 1, $black$ or $white$, etc.)
- **continuous**: continuous attributes can take infinite number of values such as real numbers or (idealized) floating point values.

Besides univariate data objects of one type, data records can also have multivariate attributes consisting of either a fixed set or a varying set of attributes. In multivariate attributes, data fields can be of the same data type (e.g. in records of geographical position containing x, y, and z) but it is possible that each data field can take an individual type of data (e.g. ID in connection with a binary attribute).

Classification does not address only individual data items. Also collections of data records show up different characteristics that are mapped to different types of data sets. Numerous types of data sets exists as there exists many types of attributes, but Tan et al. highlight three main **types of data sets**:

(1.) **Record Data** defines a collection of data records with no explicit relationship among the individual data entities. If a data record contains multivariate attributes, the collection of record data can be represented by a matrix of size m x n with m defining the dimension of the vector representing a data record (m being the number of attributes) and n the number of recorded data items.

(2.) **Graph-based Data** are data sets that contain relationships among objects in such a manner that the data objects and their relationships can be represented by a graph. The data objects are thereby the graph nodes whereas the relationship between the objects is represented by the edges between the nodes.

(3.) **Ordered Data** defines data sets with relationship in the values established by order in time or space. Ordered data can be distinguished into:
   - **sequential data** are data where each data record is associated with a time stamp
   - **sequence data** are data sets that build up a sequence of data entities in an ordered manner but without containing time stamps
   - **time series data** are collections where each data record is a time series
   - **spatial data** contain spatial attributes (such as geographical location)

For all types of data sets some common characteristics have to be considered as well. The first aspect is the **dimensionality** of the data set, as it strongly depends on the application of how many data attributes per record as well as data records in a set can be stored. Often, anomaly detection techniques have to cope with huge data amounts with high dimensionality. The second aspect to be considered is **sparsity**. Sparsity mainly is concerned the question of relevant data in order to save computation time and space. In some applications, in might be useful not to store each data record, but extract only the relevant ones. A further aspect to be considered is the resolution of the recorded data that is mainly responsible for determining the ability of the approach to identify deviations. If the scale of data items will not allow to detect distinctive information, the values may become useless.

The last factor that has impact on the nature of data is its quality. In terms of precision, this means that data recorded may be exposed to noise, collection errors, non-accuracy, missing values or inconsistency. Furthermore, recorded data may require preprocessing such as normalization or standardization, dimensionally reduction, feature subset selection, creation or extraction, discretization or binarization or some further specific transformation in order to serve as input data for an according anomaly detection technique.

2. **Labels**

   Labels are associated with data instances to denote whether a data instance belongs to a *normal* or *anomalous* data set. Accurate data instances of *normal* behavior are easier to be determined and obtained than *anomalous* data instances as it is typically difficult to specify (a complete assortment of all) possible anomalies that may arise. Anomaly detection systems can operate in different modes due to the availability of accurately labeled data. (These modes are differentiated into *Supervised anomaly detection*, *Sem-supervised anomaly detection* and *Unsupervised anomaly detection* presented later in this section.)

3. **Anomaly Type**

   The most important aspect is to verify the nature of the desired anomaly that has to be detected by the implemented anomaly detection algorithm. Chandola et al. define three different categories:

   - *Point Anomaly*: *Point Anomalies* are assumed in approaches where individual data instances are examined separately. If a data instance is considered to be different from the rest of (normal) data, the anomaly caused by this data instance is termed *Point Anomaly*.
   - *Collective Anomaly*: *Collective Anomalies* are assumed in approaches that examine collections of data instances with respect to the rest of data. The data instances may not be anomalies by themselves, but their aggregation is considered as anomalous. *Collective Anomalies* can only occur in data sets that are related to each other like sequence data.
   - *Contextual Anomaly*: *Contextual Anomalies* are assumed in approaches that examine data instances in a specific context. A data instance is considered to be anomalous only if the context conditions are present (if not, then the data instance is of *normal* data). Such an anomaly is then termed *Contextual Anomaly* (or *Conditional Anomaly* as referred by Song et al. in 2007 [84], see [34]). Of course, the notion of context has to be specified as a part of the problem formulation. Two sets of attributes are required to be collected for anomaly detection: *Behavioral attributes* and *Contextual attributes*. Behavioral attributes are those that are collected to specify the pure behavior while the contextual attributes are used to determine the context of the behavioral instance.

     Examples for *contextual attributes* are *spatial data* that define the location of data instances, edges in graphs as contextual attributes to indicate connections between nodes (containing behavior attributes), position of data instances in sequential data or time-series data, time stamps of events on sequential data or even profile data to segment or cluster data attributes into data components having the same contextual attribute.

Contextual Anomalies can be determined on data points as well as data collections so that approaches examining *Contextual Anomalies* can be related to *Point Anomalies* or *Collective Anomalies*.

4. **Output**

   Analysis of data results in an output of the anomaly detection technique. Chandola et al. distinguish between two types of output: first, by using *(binary) labels* that assign a test instance either to be *normal* or *anomalous*, or second, by assigning a *score* to each test instance that indicates the degree of this test instance to be *anomalous*. The labeled output delivers a clear categorization of test instances (being *normal* or *anomalous*) while the scoring-based approach also allows grey zones.

These four aspects allow to formulate and classify the anomaly detection problem precisely. From the specification of each of the aspects, according requirements arise and are set on the approach dedicated to solve the formulated anomaly detection issue. These requirements can be mapped on features and properties that classify anomaly detection techniques.

2.2.2 Design Decisions

The task of problem formulation requires an in-depth examination of the dedicated application (domain). The specification of problem formulation clarifies some basic characteristics (nature of data, data labels, type of anomaly and output), but it automatically yields into requirements that in turn result in features and properties of anomaly detection to be realized by the applied approach. Most of the decisions of implementing a feature or a property must be made at design time. In this section, we discuss all design time decisions such as detection principles and operational aspects [23] that must be clarified before implementing an anomaly detection approach.

The basic property of any anomaly detection system derived from definition is its operation on a system under execution. As a first design decision, Axelsson [23] distinguishes between two different detection principles:

1. **Programmed**

   The system is *programmed* to detect anomalous data if the decision between *normal* and *anomalous* is determined by a user or a function configuring the system. This programmed configuration can be done by setting the parameters of a descriptive statistic for the normal profile (e.g. number of unsuccessful logins, number of network connections) or by user/system-defined thresholds that determine the boundary between *normal* and *anomalous*. Another sort of *programmed* detection is based on the so called *default deny* policy. Here, explicitly defined benign and legal system states deal as a basis for detection as any deviations from these states are by default declared as anomalous.

2. **Self-learning**

   Self-learning detection systems learn what constitutes normal by means of sample data. Usually, typical observation data is exploited to establish a model of normal system behavior. These models can be rule-based models, descriptive statistics or even more complex structures such as Hidden Markov Models [10] or Artificial Neural Networks [6].

In this thesis, we are mainly interested in such anomaly detection systems that are not *programmed* but rather *self-learning*. The lifecycle of such anomaly detection systems consists of two phases: *training phase* and *operational phase*. In the training phase, the system is executed within a *safe* and *reliable* environment in order to guarantee normal system execution. In this benign environment, the anomaly detection system is fed with selected operational data called *training data* in order to establish the model of *normal profile* of the system operation. Different types of training data exist that determine the mode of the anomaly detection technique which is classified into *supervised*, *semi-supervised* or *unsupervised* [34].

- **Supervised techniques** are trained on labeled data sets that hold both classes of data: *normal* as well as *anomalous*. From this labeled data, a model is obtained that allows to establish a boundary between these two classes of data. Test data is then compared against the model to be classified as belonging either to *normal* or *anomalous* class. Normal data (and anomalous data accordingly) does not need to be of one unique class. In the supervised mode, the anomaly detection system can be fed with different (sub-)classes of normal and anomalous data.

- **Semi-supervised techniques** are trained on a labeled data set of only one class, which usually is the class of normal data. A model is build on basis of the training data to determine the normal profile. Test data is matched against that model and is declared as anomalous if the test data does not correspond with the model. Of course, semi-supervised techniques can also be trained by using only anomalous data to establish a model. However, it is very challenging for most of the applications to generate training data that covers completely every possible anomalous behavior.

- **Unsupervised techniques** do not require any labeled training data. Relying on the assumption that normal behavior occurs much more frequently than anomalous behavior, unsupervised techniques learn the behavioral model in the training phase based on unlabeled operational data. The more data is preserved for the training phase, the more robust the model may become.

The main factor for the selection of the employed training mode is governed by the availability of the training data and, associated with that, the type of training data. If training data is not available or is available but not labeled, then the applied anomaly detection technique has to be *unsupervised*. In contrary, if training data is available and labeled, then *supervised* or *semi-supervised* modes come into operation, of course, according to the availability of the type of labels.

After having established a robust model in the training phase, the anomaly detection system is intended to switch into the operation phase. Its responsibility is the analysis and evaluation of the data observed during system operation. For the operation phase, Axelsson discusses some operational aspects in [23], mainly related to Intrusion Detection Systems. Here, we refer to those operational aspects suitable to characterize anomaly detection systems that have to be considered at design time:

- **Time of detection** The anomaly detection system can either work in *real-time* (this terms is used by Axelsson in [23]) or nearly *real-time* performing online detection, or *non real-time* which is also referred as offline anomaly detection by the researchers of that community. Offline anomaly detection is triggered regularly or even periodically and runs delayed on (historical) data collected since the last execution of the evaluation. Online anomaly detection is more challenging as it requires means to examine and classify data immediately as soon as it occurs. Often, only an excerpt of the complete data set is present. However, online techniques open up the opportunity to implement prevention or direct reaction mechanism as deviations or anomalies are signalized at the time they occur, even before they can propagate. Any online anomaly detection approach can be operated in offline mode.

- **Granularity of data-processing** Axelsson [23] makes a difference between data processing in a *continuous* manner or in *batches* in regular intervals. Deriving from this definition, the granularity of data-processing is closely linked to time of detection.
- **Locus of data-collection** Observation or test data can be collected by different system sources in a *distributed* manner, or *centralized* collected by a single system entity. Furthermore, the data collection can be performed *direct* or *indirect*. Direct data collection in this context means that the data is used for evaluation in the form it has been collected. Hence, the anomaly detection can directly process the data. Indirect data collection means that the data has to be pre-processed before being able to be analyzed.
- **Locus of data-processing** Data processing can be either performed in a *centralized* manner by one single system component, or in *distributed* fashion by many different system entities. Beyond that differentiation of Axelsson [23], we distinguish between data processing that may be located *internal* as a component of the system that is under observation or *external* by an external system entity. The main difference is the access to information that could be exploited by anomaly detection systems located as an internal system entity, while the external system can only operate on those information with which it is supplied by the system itself. An external anomaly detection system however, is executed on a separate system and does not need to share resources with the system under observation. One main drawback of the external approach is the latency of access to system information.
- **Response to detected anomalies** Two types of responses on detected anomalies exist: *passive* and *active*. *Passive* response systems only generate an alert in case of a detected anomaly. The responsibility to react on the anomaly is transferred to a proper authority (such as the system administrator). In contrast to that, *active* response means that a reaction on a detected anomaly is implemented as a means of defense, mitigation prevention and maintenance of the system operation.
- **Ability to evolve** Systems can be statically configured in the training phase (in a similar way like the *programmed* detection principle) with fixed decision rules and thresholds that are not changeable at run time. This can be intended in systems with static environments. However, modern systems are more often required to dynamically change their behavior as they are executed in environments with dynamically changing circumstances. Such changing system behavior leads to an evolving *normal*. Accordingly, the anomaly detection system applied in such environments must also be able to adapt its established *normal* profile dynamically. The adaptation must be implemented in a self-learning or autonomous manner by using approaches from evolutionary computing, machine learning, or other domains that enable to implement self-x capabilities.

Many design decision have to be made during the development process and have been summarized in this section. The efficiency of the designed anomaly detection system, in turn, can only be evaluated in experiments during system operation.

2.3 Architectural Model

Historically, one of the major applications domains of anomaly detection (see Sec. 2.6) are intrusion detection systems. In a survey on research in intrusion detection systems, Axelsson [22] depicts a generic architectural model for intrusion detection systems. This model has often been cited by anomaly detection researchers, e.g. by Patcha et al. in [79], and has become a generic de-facto architecture for intrusion detection systems. Because anomaly detection is a major issue in intrusion detection system, Axelsson's architectural model can generally be applied to anomaly detection.

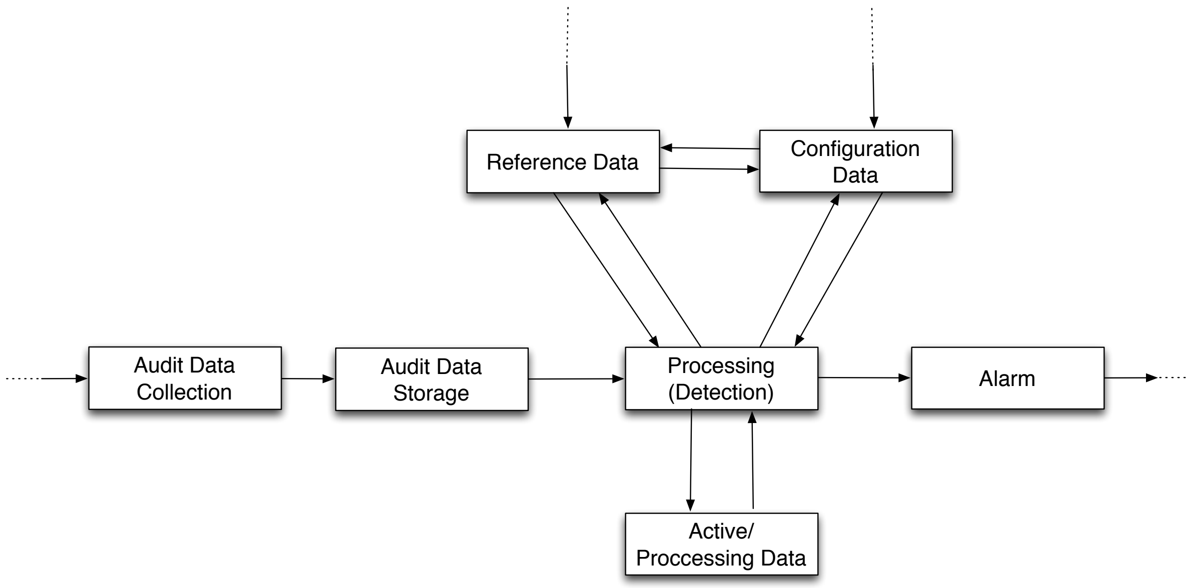


Figure 2.1: Generalized architectural model for anomaly detection (originating from intrusion detection systems [22, 79])

Fig. 2.1 illustrates the architectural model defined by Axelsson [22]. We describe the modules, the architectural model contains, mainly on the basis of [22] and [79]:

Audit Data Collection: System components or entities are monitored and produce observation data. The *Audit Data Collection* module is responsible for extracting and collecting the relevant data from the monitored entities.

Audit Data Storage: The collected audit data must be (at least temporarily) stored by the *Audit Data Storage* module. Furthermore, the storing module may encompass pre-processing to obtain a representation of data in form of an appropriate data structure for the corresponding processing module.

Processing: *Processing* is the central module of an anomaly detection system as it executes the anomaly detection algorithm or even several algorithms. It is responsible for identifying suspicious data within the stored audit data.

Configuration Data: The *Configuration Data* contains the most sensitive part of the anomaly detection system. In this module, controlling information of the anomaly detection system are defined. Besides settings such as how and when to collect audit data, it may also imply decision rules, classification boundaries and thresholds as well as rules that determine how to respond on an anomaly, etc. The *Configuration Data* mainly affects the efficiency and the performance of the anomaly detection system.

Reference Data: The *Reference Data* contains the knowledge base of the anomaly detection system. It stores *normal profiles* as well as signatures of known *anomalies*. The *Processing* module proceeds the analysis of the audit data on the basis of the knowledge base in the *Reference Data*.

Active/Processing Data: This module stores intermediate data or partial analysis results of the *Processing* module.

Alarm: This module is responsible for delivering the output of the anomaly detection system in order to initiate a response or reaction.

This architectural model apparently defines the generalized workflow for anomaly detection that is processed at system operation: collecting and storing data, analyzing the data in order to detect anomalies and trigger an alarm or a reaction if necessary.

The dotted lines in Fig. 2.1 denote the interfaces to the related system the anomaly detection system is integrated in. Monitored system entities deliver data to the collecting module of the anomaly detection system. The analysis and anomaly detection is performed in the processing module and based on the configuration data and the reference data. Even if used at operation time, this data is usually obtained before the system is set into operation and requires sophisticated examination. Furthermore, the modules holding this data provide interfaces to enable - if required - an adjustment of that data at runtime. The *Alarm* module also provides an interface as here, the output of the anomaly detection is delivered. The resulting output has to be passed to the authority responsible for initiating a response or reaction which usually is not being part of the anomaly detection system.

2.4 Data Classification

System properties are observed, collected and analyzed in order to extract information about the system state. Depending on the applied approach, the observation and analysis process is usually done in an autonomous but at least in an automatic manner. In most of the applications presented above, an anomaly detected reflects a potential threat of the system. A simple classification can be applied here which only differentiates between ***safe*** system state when matching the normal profile and ***dangerous*** system state in case an anomaly has beed detected.

However, even with this simple classification approach, it is important to consider that the boundaries between the *normal* and the *anomaly* are not fixed. The borderline between *safe* and *dangerous* may be blurred as differentiation between *normal* and *anomalous* is often based on decision rules and thresholds. Obviously, this can lead to *false alarms*. The definition of *false alarms* is always related to the perspective and the context of the testing.

Two perspectives are possible:

1. When testing for matching *normal*, *normal* becomes *positive* while consequently the *anomaly* is considered as *negative*. Referring to this perspective, the following definitions for anomaly detection *false alarms* will result:

- **false positive** The testing data is classified as *normal*. However, the result of the test is incorrect as the test failed to recognize the anomaly. As *normal* is identified as *positive*, this classification test causes a *false positive*.
- **false negative** The testing data is classified as *anomalous*. However, the result of the test is incorrect as the test failed to identify that the testing data in fact matches *normal*. As *anomalous* is identified as *negative*, this classification test causes a *false negative*.

2. When changing the perspective and testing for *anomaly*, the *anomaly* becomes *positive* while consequently the *normal* is considered as *negative*. The specification of *false alarms* must then be interchanged:

    - **false positive** The testing data is classified as *anomalous*. However, the result of the test is incorrect as the test failed to identify that the testing data in fact matches *normal*. As *anomalous* is identified as *positive*, this classification test causes a *false positive*.
    - **false negative** The testing data is classified as *normal*. However, the result of the test is incorrect as the test failed to recognize the anomaly. As *normal* is identified as *negative*, this classification test causes a *false negative*.

Of course, false alarms degrade the performance and the reliability of the anomaly detection mechanism and are therefore undesired.

The mapping *normal* on *safe* and *anomalous* on *dangerous* is not necessarily true for all application domains. In novelty detection (see [34]) e.g., novel patterns in data are typically not assumed to be suspicious, but furthermore, detected novel patterns are integrated into the *normal profile*. Referring to the context of autonomous systems we are dealing with, behaviors can change dynamically because of a self-adaptation or self-reconfiguration of the system. Novel behaviors may have occurred that do not match a preliminarily trained *normal* even though these behaviors have been intended. In that case, the simple classification approach will not take effect and a more distinctive approach for classification is required here.

In [64], Kumar et al. present such a classification approach as a premise for intrusion detection systems. We apply the differentiation proposed by Kumar et al. for the classification of behaviors in our anomaly detection approach. In contrast to assigning *normal* to *safe* and *anomalous* to *dangerous*, we differentiate between *normal* and *anomalous* and between *safe* and *dangerous* separately. Based on this, the following classification classes result:

Class I	: **normal** and **not suspicious**
Class II	: **anomalous** and **suspicious**
Class III	: **anomalous** and **not suspicious**
Class IV	: **normal** and **suspicious**

Class I and Class II are closely related to the simple classification approach presented above. Class I reflects normal and safe behavior while Class II identifies deviations from *normal* classified as threats. Class III outlines behaviors that do not match the *normal* but which are not suspicious. As already mentioned above, this classification class is essential for changing system behavior due to self-adaptations or self-reconfigurations. Obviously, the resulting behavior will possibly not match *normal*. But as resulting from an autonomous system change, this previously unknown behavior should not be classified as a potential threat. Class IV illustrates the case of behavior matching the *normal profile* which comprises potential threats. This situation may be caused by various scenarios. It can simply be a *false alarm*. On the other hand, this can happen if *normal* and *safe* has not been defined accurately. Beyond that, learning capabilities may be applied that lead to continuously updating of normal behavior with potentially suspicious behavior of threats invading in a creeping manner. It is also possible that changes in the environment will not be registered properly. Unsafe system states may result if the behavior maintains unchanged.

Our anomaly detection approach is dedicated for self-reconfiguring systems that are applied in dynamic environments. For such applications, the simple classification approach is not sufficient. Therefore, we will apply the defined classification classes which have been inspired by Kumar et al. ([64]).

2.5 Anomaly Detection Techniques

The broad spectrum of application domains and anomaly detection problems is presented in section 2.6. As the nature of attributes determines the applicability of an anomaly detection technique, the variety of input data types and types of anomalies to be detected related to different domains and problems lead to the requirement for appropriate approaches and justifies the need for an equivalent amount of different anomaly detection techniques. In this chapter, we categorize the existing techniques and introduce them in a short manner.

Tan et al. provide an overview on anomaly detection techniques in [87, Chapter 10]. They can be classified into *Supervised*, *Semi-Supervised* or *Unsupervised* as depicted in section 2.2.2. Additionally, they differ in the number of attributes and the relation of values to define an anomaly, in the perspective of detecting global or local anomalies, in the degree to which a point is an anomaly, in the ability of identifying one anomaly at a time versus many anomalies at once, in their evaluation effectiveness as well as in their (computational) efficiency. Often, the individual techniques discussed in this section have been developed to serve specific application domain problems and tend more to be specialized instead of being generic. However, referring to [87], anomaly detection techniques overlap in problem solving characteristics so that they be grouped at high-level into:

1. **Model-based Techniques**

   These techniques first build a model out of the provided data. Such a model can be e.g. a statistical model for distribution, or probability of distribution, a clustering model as well as regression model. Model-based techniques distinguish between two different classes of data: *normal* is every data record that fits the model, and *anomalous* is data that does not fit the model. For model construction, a training data set is obligatorily required as it becomes very difficult to construct a model if no training data or no statistical distribution is available. The obtained models then tend to be highly specialized for the particular application.

2. **Proximity-based Techniques**

   These techniques define a proximity measure for distinction between *normal* and *anomalous*. An anomaly in this context is an data object that is far from most of the other objects with respect to the proximity measure. A subclass of proximity-based techniques is represented by distance-based outlier detection techniques.

3. **Density-based Techniques**

   These techniques comprise two different views to classify data points as anomalies: if data points are in regions of low density relatively distant from their neighbors, or if their local density is significantly less than the density of most of their neighbors. These approaches also rely on proximity measures to some extend.

All these high-level technique categories can be reflected in approaches implemented by the particular anomaly detection techniques that originally rely on concepts adopted from statistics, machine learning, data mining, information theory, spectral techniques, etc. that now can be considered as related disciplines. The latter of this section provides a collection of anomaly detection techniques with each including a basic introduction mainly related to Chandola et al. [34] and Tan et al. [87].

2.5.1 Classification-based Anomaly Detection

Classification-based approaches come from machine learning and data mining domain. Referring to [34], "A classifier that can distinguish between normal and anomalous classes can be learnt in the given feature space." Following from this assumption, classification-based approaches operate in a two-phase fashion having a training phase and a testing phase. The key issue is to learn classifiers (similar to model) from a set of labeled data instances (in the training phase) and classify test instances according to the learnt classifier (in the testing phase). With respect to the number of labels, classification-based anomaly detection techniques can be grouped into *multi-class* anomaly detection, if multiple normal classes exist and consequently, multiple labels are available for *normal* data instances, and *one-class* anomaly detection techniques, if all training instances have only one class label. Anomalies are identified in *multi-class* anomaly detection if a test instances cannot be classified as normal by any of the classifiers, each associated to one *normal class*, whereas anomalous data instances in *one-class* anomaly detection are identified based on a learnt discriminative boundary. Different approaches exists to build classifiers and are exploit for anomaly detection:

1. **Neural Network-based Anomaly Detection**

   Neural networks are computational models inspired by the central nervous system (brain) applied in machine learning and other related fields (see Wikipedia: [6]). Neural networks are capable of machine learning and pattern matching by computing input data through hidden layers to so called output neurons. A neural network consists of nodes and interconnections between nodes that usually contain adaptive weights which are continuously updated in order to realize the learning algorithm that trains the neural network based on the input data.

   Usually, neural networks implement non-linear functions in the hidden layers. They are applied to compute input values in a parallel and distributed manner in order to solve problems that are hard to be solved by ordinary methods.

   Applied in anomaly detection, they can cope with one-class anomaly detection problems as well as multi-class. The training phase is used to build a stable classifier represented by the network. If the neural networks accepts a test input, then the test instance is classified as normal. If the neural network rejects a test input, it is classified as anomalous.

Chandola et al. list some examples for anomaly detection using different variants of neural networks, such as Multi Layered Perceptrons, Neural Trees, Radian Basis Function Based, Adaptive Resonance Theory based, Replicator Neural Network, etc. in [34].

2. **Bayesian Network-based Anomaly Detection**

   Bayesian networks are probabilistic graphical models to represent (often causal) relationships between variables (represented by nodes) in the Bayesian sense (see Wikipedia: [7]). The nodes are associated with probability functions giving the probability of the variable represented by the node. The edges between the nodes represent conditional interdependencies of nodes.

   Bayesian networks are used for anomaly detection with multi-class properties. Basically, for a given input data (which can be univariate categorical data as well as a multivariate categorical data set with aggregated per-attribute values), the Baysian network estimates the posterior probability of being of a particular class. [34] provides some references to examples that use variants of Bayesian networks for anomaly detection.

3. **Support Vector Machine-based Anomaly Detection**

   Support vector machines (SVMs) come from the domain of machine learning and are supervised learning models with associated learning algorithms (see Wikipedia: [17]). With their learning algorithms, SVMs are able to analyze data, recognize patterns, perform classifications and regression analysis. SVM require input data in the form of points in space of both classes (normal and anomalous) of data for training in order to establish a non-probabilistic binary linear classifier. By using Kernel methods (see Wikipedia [12]), SVMs can also perform non-linear classification by implicitly mapping the inputs into high-dimensional feature spaces.

   Based on its classification method, SVMs can be applied to one-class anomaly detection. The learnt classifier builds the classification boundary (either linear or a more complex non-linear boundary) between the exclusive regions . A test instance is classified according to the region it falls in.

   Chandola et al. provide some further references to variants of SVMs and they mention some example implementations for anomaly detection with SVMs in [34].

4. **Rule-based Anomaly Detection**

   Rule-based systems rely on rule-learning algorithms to determine rules that capture the normal system behavior in their training phase. Rule-based anomaly detection is able to cope with one-class and multi-class properties. The key issue in rule-based anomaly detection is formed by a confidence value that is associated with each rule determining a proportional value to the ratio between the number of correctly identified training instances and the total number of training instances covered by that rule. Anomalies are those input instances that are not covered by any of the learnt rules. Each test instance is assigned an anomaly score based on the inverse of the confidence value associated with the rule by which the test instance is captured best. Again, Chandola et al. have collected in [34] some references to variants of rule-based techniques as well as some application examples.

Different techniques exist for classification-based anomaly detection that address different applications domains. Some of the (for this thesis) relevant examples referenced by Chandola et al. are discussed in Chapter 4. However, most of the existing approaches predominantly solve *Point Anomaly* detection problems.

Summing up, classification-based techniques can use powerful algorithms, especially for multi-class classifications problems, in order to discriminate between instances of different classes. The computational complexity of the respective technique depends on the complexity of the applied classification algorithm. However, most of the techniques produce high computational costs in the training phase when learning the classifier while the testing of input instances is usually simple and fast. The quality of the classification results relies on the availability of accurate trainings sets and their coverage.

2.5.2 Nearest Neighbor-based Anomaly Detection

Nearest neighbor-based anomaly detection follows the assumption, that "normal data instances occur in dense neighborhoods, while anomalies occur far from their closest neighbors." (see [34]). In order to determine whether data instances are close or far from other data instances, nearest neighbor-based approaches require a distance or a similarity measure. Computing this measure can be done by different methods depending on the nature of input data (univariate or multivariate, categorical, continuous, etc.) as well as on the type of output expected. Mainly, nearest neighbor-based anomaly detection techniques can be categorized into distance-based and densitiy-based techniques:

1. **Distance-based Anomaly Detection**

   Distance-based techniques are also referred as *Proximity-based Outlier Detection* by Tan et al. in [87]. Basically, they determine an anomaly score, or outlier score respectively, for each data instance defined as the distances to its k^{th} nearest neighbor among the data set. There are mainly two different methods applied to declare data instances as anomalous. On the one hand, a threshold is defined on the anomaly score to set a boundary between data that is still classified as *normal* and such data instances that are treated as *anomalous*. On the other hand, anomalies are those data instances that belong to the set with the n highest anomaly scores. Furthermore, various approaches exist to calculate the anomaly score of a data instance. One example for this is counting the number of neighbors within d being a distance value (radius in hyper-sphere). Chandola et al. discuss in their article several methods for calculating distance measures for the various data types and provide many references to corresponding approaches and publications. As these approaches vary (from calculating Euclidean distances up to more dimensional hyper-grid representations), their computational complexity obviously depends on the complexity of the applied approach.

2. **Density-based Anomaly Detection**

   Density-based anomaly detection techniques consider for each individual data instance the density within its neighborhood. As a basis, density-based anomaly detection techniques follow the simple principle to test whether a data instance is lying within a neighborhood of low density (meaning that there are very few other data instances close to it). In such a case, the data instance is declared as *anomalous* whereas in case of a dense neighborhood, the data instance is put to the class of *normal* data. The neighborhood is thereby determined by distance (radius) or by restricting it to the k^th nearest neighbors.

However, if data sets form regions (or so called clusters) with different densities, this general assumption may lead to false results so that this basic technique shows up poor performance. In order to prevent such failures in anomaly detection, various improvements and extensions of this basic principle have been proposed in literature, which has been collected and summarized by Chandola et al. One example is to refer to a local relative densitiy such as *Local Outlier Factor (LOF)* (see Wikipedia [13]) or variants of it in order to classify data instances.

Besides a spatial distance, there are approaches that use similarity measures. Similarity-based techniques are in particular applied in anomaly detection problems concerning sequence data in order to find the nearest neighbor for a given data sequence.

Nearest neighbor-based techniques are usually unsupervised anomaly detection techniques as they do not rely on any predetermined assumptions regarding the input data (e.g. of data distribution) but are purely driven by the data produced by the application. An accurate set of data instances is required which adequately covers the possible data classes. For some applications, semi-supervised techniques are applied if only *normal* data sets are available for training.

Concerning the performance, nearest neighbor-based techniques require $O(N^2)$ (N is the data size) computational complexity to classify data instances as they rely on measures reflecting the (pairwise) interrelation between data instances. Of course, some optimizations in terms of complexity and efficiency have been proposed by different researchers which are referenced by Chandola et al. in [34]. However, in those applications where an anomaly score is required for each test instance, the computational effort is huge so that they seem not applicable for online methods. On the other hand, optimizing approaches that reduce complexity by classifying only a subset of test instances may lead to inaccurate and incorrect anomaly detection.

In fact, the performance and the complexity mainly depends on the distance or density measure applied as well as on the nature of data (concerning the number of attributes) provided by the application domain. Furthermore, nearest neighbor-bases techniques are sensitive to the choice of the parameters and therefore, require detailed investigation for finding appropriate measures, setting a well-defined value for k and defining according thresholds and boundaries which altogether makes their design to be very challenging.

2.5.3 Clustering-Based Anomaly Detection

Clustering is defined as the task of finding groups of strongly related objects out of given data instances (see Wikipedia [8]). A cluster builds up a particular kind of model so that clustering-based techniques are part of the class of model-based anomaly detection. Various different clustering algorithms exist and, in general, they all can be applied to anomaly detection. When applied to anomaly detection, the clustering-based techniques can be categorized in three groups with respect to defining anomalies:

1. **Anomalies outside Clusters**

   Anomalies are considered as those data instances that do not belong to a cluster. Clustering-based anomaly detection techniques belonging to this group assume that all data instances that are part of a cluster (determined by a clustering algorithm) are *normal* *data*, while all residual data that do not belong to any cluster are declared *anomalous*. The key challenge thereby is to define by an objective function or another measure (such as *Gaussian* distance or *Mahalanobis* distance [87]) to which degree an object belongs to a cluster or not. However, such clustering algorithms are not primarily designed for detecting anomalies as their main objective is the identification of clusters. Consequently, this can affect the performance and, in particular, the efficiency of anomaly detection.

2. **Anomalies far from Clusters**

   Anomalies in this category are considered as those data instances that are far away from any cluster. Clustering algorithms such as *Self-organizing Maps*[15] or *k-means Clustering*[11] are used to find clusters for the given data instances. Then, for each data instance an anomaly score is calculated that reflects the distance of the data instance to the closest centroid of the cluster. Closely related to distance-based anomaly detection techniques, anomaly detection is based on testing each data instance and declaring *anomalies* as those data instances with an anomaly score above a predefined threshold.

3. **Anomalies represented by small and sparse Clusters**

   Clustering techniques of the first category explained above (and partially clustering techniques applied in the second category) rest upon the fact, that the applied clustering algorithm does not force data to be allocated to a cluster. However, there are also clustering algorithms that mandatorily assign each data item to one cluster. Therefore, data instances that form a cluster may also be an anomaly on the whole. According to the nature of anomaly, these data instances shall be rare and therefore, found in clusters that are small and sparse. These anomalous data sets joint to clusters represent collective anomalies. For such approaches, is it a key challenge to identify the parameters consisting of size and density that determine to which degree a cluster is classified as *normal* or *anomalous*.

Clustering-based techniques are usually unsupervised anomaly detection techniques. However, in some approaches only normal data is specified for training, so that approaches are also being explored that belong to semi-supervised anomaly detection (see [34]).

The performance and efficiency of anomaly detection using clustering-based techniques is strongly related to the clustering algorithm, which encompasses decisions about the minimum size of cluster, number of clusters, etc. Some of the techniques require distance or density computations which makes them very similar to nearest neighbor-based techniques. The main difference is how they evaluate the test instances: in clustering-based techniques data instances are tested with respect to the cluster they belong to, while the nearest neighbor-based techniques evaluate data instances in reference to their neighborhood.

The computational complexity in the training phase is basically $O(N^2)$ but in the testing phase they are very fast as each test instance only has to be matched against the (constant number of) existing clusters.

2.5.4 Statistical Anomaly Detection

Statistical anomaly detection techniques underlie a statistical (or stochastic) model for data distribution. They presume *normal* data to lie in regions of high probability of the applied model whereas anomalies occur in low probability regions of the according model. Statistical anomaly detection technique are differentiated into parametric and non-parametric techniques:

1. **Parametric Techniques**

   Parametric techniques require the definition of underlying (parametric) distribution by which Θ represents the distribution parameters which are derived on the basis of a given training *normal* data set. Concerning anomaly detection, parametric techniques apply two different options to define anomaly scores for observation test data: First, the distribution is associated with a probability density function $f(X, \Theta)$ for X being the observation test data. The anomaly score of X is defined by the inverse of the probability density function f. On the other hand, parametric techniques use statistical hypothesis testing [16] by examining whether the observation test data X is covered by the estimated distribution. If X is not covered by the distribution, a test statistic delivers a probabilistic anomaly score for the test data X.

   Parametric techniques can be categorized according to the type of distribution. Typical types for distributions are

   I. **Gaussian-based Distribution**: The data is assumed to appear according to Gaussian (or Normal) distribution which is the distribution model that is most frequently used. Relying on two parameters, mean μ and standard deviation σ, a probability can be calculated for each test data. The distance of the test data to the mean defines the anomaly score of the test instance which in turn identifies the test instance being *normal* or *anomalous* based on thresholds set for the anomaly score. Threshold settings are strongly application-specific. Furthermore, as data can be univariate or multivariate, different techniques can be applied to distance calculation and threshold setting.

   II. **Regression-based Distribution**: Regression models are prediction methods and have been applied to anomaly detection in time series data. The test data is matched against the predefined regression model. The residual of the test instance is the part not matching the regression model. This residual part is used to determine the anomaly score calculated either on the basis of the magnitude of the residual or on statistical tests. There are so called *robust regression* methods which belong to the class of those methods being able to cope with anomalies in training data. Well known robust anomaly detection techniques for time series data are *Autoregressive Integrated Moving Average* (ARIMA) and (for multivariate data) *Autoregressive Moving Average* (ARMA) which are explained in short in [5].

   III. **Mixture-based Distribution**: In mixture-based techniques, the data is modeled based on a mixture of parametric statistical distribution functions. Basically, two approaches for mixture-based distribution exist that are based on different assumptions: in the first one, the mixture is associated with the fact that normal data is considered to be modeled by one statistical distribution while anomalous data is modeled by another clearly separated distribution where the main task in the testing phase is to verify to which of the both distribution the test data belongs to. The second approach models only normal data by using a mixture of statistical distribution functions. A test instance that does comply with any of the applied distributions is declared as *anomaly*. Mixture-based distribution approaches are potentially powerful as they allow a more realistic model of the data set. However, the models reflecting the reality approximately exactly tend to be complicated to understand as well as to use them.

2. **Non-Parametric Techniques**

   For non-parametric techniques the underlying distribution is assumed to be unknown a priori. The statistical distribution model is derived from the data set autonomously.

   I. **Histogram-based Anomaly Detection**: The most common non-parametric technique constructs histograms to maintain profiles for data. The histogram is build up based on *normal data*. Often, histogram-based approaches are referred to as frequency-based or counting-based approaches, as the area representing a data attribute reflects its frequency in the training set. The key challenge when constructing the histogram is the definition of the size of a bin (interval for attribute value) as this is the basis for evaluation of test data. A inadequate choice of the bins causes the key source for producing false alarms.

      Histograms are applied to univariate data as only one dimension is illustrated in the histogram. In multivariate problems, multiple histograms are generated, one for each attribute, which deliver per-attribute anomaly scores for test data instances that finally can be aggregated to determine the anomaly score for the entire data instance.

   II. **Kernel Function-based Anomaly Detection**: These techniques are usually semi-supervised anomaly detection techniques. They use a kernel function[12] to estimate the actual density in order to determine the *probability density function (pdf)* of the given data set. All test instances that lie in a low probability region of the estimated distribution are identified as *anomalies*.

Statistical models are powerful in anomaly detection as problems and, accordingly, data sets of problems can be adequately represented by a statistical model reflecting a good approximation of reality. The key challenge of statistical based anomaly detection approaches is to identify the right statistical model that reflects the normal data correctly. Wrong choice of the model or its parameters will lead to false evaluation of test instances. Most of the approaches are dealing with *Point Anomalies* as the objective of such approaches is to calculate an anomaly score for an individual test instance. The complexity of statistical anomaly detection primary depends on the fact whether the statistical model is known in advance or has to be estimated. Secondly, the complexity is also related to the statistical model that is applied as well as the number of considered attributes. In most of the approaches, the testing phase can be performed in constant or linear time as the test instances has only to be matched against the fitting to the statistical model. However, applying more complicated approaches, such as kernel functions, the testing phase can also have quadratic time complexity.

2.5.5 Information Theoretic Anomaly Detection

Information theoretic anomaly detection techniques are based on the assumption that irregularities in information content of data sets are traced back to anomalies. Therefore, an analysis of the information content is required and usually performed by applying measures from information theory like *Kolomogorov complexity*, entropy, relative entropy, uncertainty, etc. Because of the information theoretic measures, these anomaly detection techniques can be classified as unsupervised (no examples or training data for anomalous data are required). Anomaly detection applying information theoretic approaches always consider the entire data set D for which $C(D)$ denotes its complexity (such as *Kolomogorov complexity*, entropy, relative entropy, uncertainty, etc). They examine every possible subset in order to determine a minimal subset I for which the distance of complexities denoted by $C(D) - C(D - I)$ is maximized. The subset I then contains the instances that are declared as anomalies.

Information theoretic techniques are also applied to ordered data sets such as sequential data with the objective to find the substructure I such that $C(D) - C(D - I)$ is maximized. As the sequential data is split into subsequences, it is the key challenge to determine the size of the subsequences (window) that is optimal to anomaly detection in that particular data set.

Considering the entire data set with all possible subsets leads to a basic computational complexity of exponential time even though some optimizations and approximations exists that are listed by Chandola et al. Basically, the performance of the anomaly detection approach is highly dependend on the choice of the information theoretic measure that should be sensitive enough to enable the aspired anomaly detection. Additionally, the performance of anomaly detection in sequential data using information theoretic measures is strongly related to the quality of the choice of the subsequence window size.

2.5.6 Techniques for Collective Anomaly Detection

Collective anomaly detection techniques are discussed by Chandola et al. in [34] only in short. Collective anomalies are identified by a co-occurrence of data instances in such a way that this co-occurrence does not match normal behavior (see Section 2.2.1). This incorporates that there exists a relationship among the data instances. Three types of data relationships reflect the most frequent collective anomaly detection problems and have been identified by Chandola et al.: *sequential*, *spatial* and *graph* relationship.

Anomaly detection in *sequential data* aims to detect anomalous subsequences or entire sequences in event sequence data (sets) or time-series data (sets). As this type of techniques is the most interesting one for the problem addressed in this thesis, a detailed discussion is provided in Section 2.5.8.

For *spatial and graph data*, anomaly detection has not become a meaningful research topic yet, so that only little research has been done addressing these problems. Chandola et al. only present the basic principles without going into details of the applied techniques: Spatial anomaly detection techniques aim to detect connected regions in spatial data that show up anomalies. The data set is segmented by use of appropriate segmentation techniques. The resulting segmented data subsets are explored to find collective anomalies by e.g. considering the anomaly probability of each data instance within the region (which in fact is detection contextual anomalies) and connecting all anomalous data instances into the spatial structure. Typical problems for spatial anomaly detection are addressed by image processing.

Graph relationships between data lead to graph anomaly detection techniques that aim to find anomalous subgraphs within the data. Different assumptions on anomalies are considered. On the one hand, anomaly detection involves the classification of a sub-graph with respect to the entire graph. Thereby, measures like regularity of graph or its entropy are important ingredients. On the other hand, anomaly is related to the frequency of sub-graphs within a large graph by taking into account the size of the sub-graph in order to determine anomaly scores for the particular sub-graphs.

Detecting collective anomalies is more challenging than detecting point anomalies as it requires to examine the structure in data which is strongly related to the type of the relationship between the data.

2.5.7 Techniques for Contextual Anomaly Detection

The techniques discussed in the sections above concentrate on either detecting *point anomalies* or *collective anomalies*. Research in techniques for *contextual anomalies* (see description in Section 2.2.1) often relies on reducing the problem of detecting *contextual anomalies* either to the problem of *point anomalies* or *collective anomalies* .

1. **Contextual Point Anomaly Detection**

   A point anomaly is an anomaly in an individual data instance. Contextual point anomalies are anomalies in individual data instances with respect to its context. The context of each test data instances is determined by its *contextual attributes*.

   For anomaly detection of contextual point anomalies, it is possible to use a common technique for point anomalies which is appropriate to the considered problem (as those presented in the sections above) in correlation with the according *contextual attributes*. A data instance d is represented, for example, as $[x, y]$ with x being the behavioral attributes and y being the contextual attributes. Both, behavioral attributes and the contextual attributes, are respresented by data models (e.g. statistical distribution models), but each by an individual one. The anomaly score for the data instance d is calculated on the basis of the probability that the behavioral attributes x are generated when the contextual attributes y are present.

2. **Contextual Collective Anomaly Detection** *Collective Anomalies* occur in data structures such as time-series or sequence data in which not one individual data item is anomalous but the entire structure. Typically, prediction-based techniques are applied here, as usually, the future behavior is expected to be estimated based on observations of behavior history.

   The behavioral structure is learnt in the training phase and constructs a prediction model for expected behavior within the given context. To realize this, regression analysis techniques [14] offer powerful methods for behavior prediction as they model and estimate relationships among variables. Applying regression, contextual attributes are used to predict the behavioral attributes. Numerous variations of regression analysis have been examined and proposed contextual anomaly detection for time series or sequential data. These different approaches are references by Chandola et al, in particular, ARMA and ARIMA models [5] specific to contextual anomaly detection in time-series, to name some of them.

Usually, problems of contextual anomaly detection are reduced to either problems of point anomalies or collective anomalies so that the according techniques are applied, extended by taking the data context into account. Because of this reduction, the complexity of the anomaly detection technique is related to the reduction technique as well as the applied anomaly detection technique. Considering the context of data (either data instance or data structure) enables to detect a class of anomalies that might not have been detected by the *pure* anomaly detection technique (without the extension of context information).

2.5.8 Sequence Data Anomaly Detection

Sequence data is an ordered series of data instances. Many applications produce data sequences or time series data. In his PhD thesis [33], Chandola emphasizes that "anomalies in sequences can only be detected by analyzing data instances together as a sequences" so that different anomaly detection techniques are required as the traditional once discussed above.

In [34], only a brief overview is given on sequence data anomaly detection, but Chandola et al. provide in [35] a more comprehensive survey on anomaly detection for sequence data, and in particular, in this article they concentrate on *symbolic* (or as they refer to *discrete* data). This section mainly summarizes the techniques described in [35].

Many different approaches for techniques for sequence data anomaly detection exist because it is a very common problem in a wide range of domains. Many applications produce sequentially ordered data, but the nature of data varies and is related to the application domain. Chandola et al. mainly distinguish between symbolic or discrete data and continuous or time-series data (time-series of symbolic data belongs to the class of symbolic or discrete data). Sequential data can, of course, consist of univariate as well as multivariate data instances. Besides the nature of the data, the nature of anomaly to be detected may be different. Several techniques have been proposed for symbolic sequences within specific domains, but, up to now, it is not well-understood and has not been examined in detail to what extend a technique developed for a particular application domain would perform in a completely different domain.

Chandola et al. differentiate the problem of anomaly detection of sequential data into three different assumptions on anomalies that refer to different classes of anomaly detection:

1. Sequence-based anomaly detection
2. Continuos subsequence-based anomaly detection
3. Pattern frequency-based anomaly detection

These three classes of sequence data anomaly detection have different characteristics, problem formulations and hence, they apply different techniques. In the following, basic anomaly detection techniques are presented for each class of sequence data anomaly detection, but these techniques are mainly related to *symbolic* sequence data (according to the survey offered by Chandola et al. [35]):

1. **Sequence-based anomaly detection**

   This class of anomaly detection is concerned with detecting entire anomalous sequences in a set of sequences with respect to the defined or trained normal. Existing techniques operate either in semi-supervised mode with available labeled training data or in unsupervised mode in case of unlabeled data.

   Assuming, that all sequences are of equal length, a sequence can be treated as a data record having n features (n is the length of the sequence). Then, the detection of anomalous sequences is related to the problem of point anomalies in which the test data instance is composed of multiple feature values so that any appropriate point-based anomaly detection technique (fitting the application and the problem as described in the sections above) can be applied here, e.g. similarity or distance-based techniques.

   However, in practice, sequence-based anomaly detection is facing the key challenge of varying sequence lengths of individual sequences and, referring to this, the challenge of sequences that may not be aligned with each other and especially, with the set of normal data. In [53], Gusfield explores established methods for sequence alignment and sequence matching for sequences and strings. These methods offer potentials to be applicable to sequence-based anomaly detection as the the sequences addressed here are symbolic sequences that can be referred as *Strings*.

To deal with sequences of different lengths, two different approaches exists:

(a) transforming the sequences with different lengths into sequences of equal lengths which leads to the fact that again, point-based anomaly detection techniques can be applied here. However, the transformation can only be applied if sequences can be property aligned

(b) use techniques that can operate on sequences of unequal lengths. This approach is chosen if the original sequence is required to be maintained in length and structure or if alignment of sequences is infeasible or of unacceptable effort.

According to [35], the latter can be classified in four different categories of techniques:

1.1 **Similarity-based Techniques**

In similarity-based techniques, sequences are compared pairwise by applying a specific similarity or distance measure (k-nearest neighbor, clustering-based techniques etc.) for anomaly score calculation. Other approaches are based on counting the number of positions in which the two sequences match. However, these approaches require sequences of (nearly) equal lengths. For sequences with different lengths, methods are used to compute the *longest common subsequence* as a similarity measure. Similarity-based techniques by nature do not consider relations between different pairs of symbols within a sequence. A huge selection of approaches exists and can be found in the literature. If the problem considered can be transformed to a problem of similarity (or distance), then it can be solved by one of the many existing similarity-based technique. Obviously, the performance of the anomaly detection is depending on characteristics and parameters of the applied approach as well as on its suitability to the problem. Similarity-based techniques suffer on the complexity as pairwise comparison is of $O(N^2)$.

1.2 **Window-based Techniques**

Window-based techniques are usually applied as semi-supervised methods and hence are based on available training data. Mainly, they address the problem of assuming that causes for an anomalous sequence can be located within subsequences of that sequence and assuming that those anomalies - when analyzing the sequences as a whole - either itself or their cause might not become detected. The basis of these techniques is that sequences are split into fixed-sized (overlapping) windows of length k. Thereby the windows are sliding by one symbol per window along the sequence. In the training phase, for each window extracted from the training sequences the frequency of occurrence is determined. In the testing phase, each of the windows of the test sequences is analyzed separately with respect to a trained normal obtained from training sets. Based on this, the analysis of a window results in an anomaly score (e.g. the inverse of the frequency) that is assigned to each window. The individual anomaly scores of the windows belonging to one sequences are aggregated to result in an anomaly score for the entire sequence. If the anomaly score of a sequence exceeds a predefined threshold the sequence is declared as anomalous (normal otherwise). Different variants exist for defining the window's anomaly score as well as for calculating the aggregated anomaly score of the associated sequence. The basics of this variants are explained in short in [35]. Besides the basic examination of frequencies of subsequences, there are techniques that make use of so called *lookahead pairs* defined as $< \alpha, \beta >_j$ with β following α at the jth position in the window. Thereby, the number of mismatches of these expected lookahead pairs contributes to the anomaly score calculation. Others use measures like *Hamming Distance* (such as Hofmeyr et al. [56]) to obtain the deviation between the test window and the windows from normal data, similarity measures (such as Lane and Brodley [65]) that take into account the location of windows within the entire sequence or classifier approaches to determine the anomaly scores of the individual windows. For all these diverse approaches, obviously, different methods exists to derive the according sequences' anomaly scores. In [35] Chandola et al. discuss the problem of merely accumulating window anomaly scores that potentially may lead to washing out anomalous regions in sequences. They give a basic introduction to methods for local combinations of window anomaly scores (such as *locality frame count* (LCF)).

Windows-based techniques are well applicable in applications that assume anomalies to be located in regions of the underlying test sequences. The biggest challenge is to appropriately choose the windows size k which strongly effects the powerfulness of a method to differentiate between normal and anomalous. Furthermore, storing all patterns occurring in windows requires huge amount of memory, in particular in case of high variations in the pattern the memory requirement can be exponential with respect to the size of the symbol's alphabet.

1.3 Markovian Techniques

The key characteristic of Markovian techniques for sequence-based anomaly detection is based upon analyzing the probability of occurrence of a symbol t_i with respect to the sequences of k symbols occurred prior to that. That means that the history of k symbols is used to determine the probability of symbol t_i in a test sequence t. In general, this probability is defined as $P(t_i|t_{i-k} \dots t_{i-1})$. The overall probability of a given sequence t is established using chained rules (see [35]), such as by multiplying the individual probabilities $P(t) = \prod_{i=1}^{l_t} P(t_i|t_1 \dots t_{i-1})$ (with l_t length of l). Markovian techniques are semi-supervised anomaly detection techniques that learn parameters and establish a probability distribution model for the occurrence of the symbols in their training phase.

Considering the length of history of a sequence k analyzed by the anomaly detection technique, Chandola et al. in [35] distinguish between three different approaches for Markovian techniques: *fixed Markovian techniques*, *variable Markovian techniques* and *sparse Markovian techniques*.

Fixed Markovian techniques use a fixed value for k. In this class of techniques, the probability of a symbol in the test sequence is determined by a fixed length of history of k symbols. $k = 1$ thereby forms a special case where the probability of a symbol is only related on its direct predecessor which has gained special attention in the research community because of its simplicity (see further references provided by [35]). Having established all probabilities of symbols with respect to their history, all these transitions have to be stored. Typical data structures for storing the probabilities (or frequencies respectively) are *Extended Finite State Automata* (EFSA) [9]. The size of the EFSA is depending on the parameter *k*, the size of the alphabet - which is determined by the application, but in particular on the variations of sequences defined as training data. Obviously, the constructed EFSA can be of huge complexity.

Fixed lengths of history may lead to unreliable estimation of probabilities for specific cases (e.g. if a training sequence of length *k* occurs only once in the training set). Variable Markovian techniques address this problem by introducing variable length of history for probability estimation. To compute the optimal value for *k* for a training set, models such as *Probabilistic Suffix Trees* (PST) are applied. Based on predefined thresholds, it is ensured that only such sequences are considered which in the training set occur frequently enough. Each of the (sub)sequences is pruned to an individual length in the PST.

While fixed-length and variable-length techniques require a contiguous and immediately preceding symbols for analysis, sparse Markovian techniques are more flexible as they allow symbols to be replaced by wildcards within sequences. Hence, the probability of occurrence of symbols is based on a sparse history. These techniques use so called *Sparse Markov Transducers* (SMT) which are similar to PST to estimate the probability of symbols conditioned on a preceding sequence. In the training phase, a collection of different variations of wildcard positions is constructed. Finding an adequate position for a wildcard symbol is a challenging issue and leads to a complex training phase.

Markovian techniques analyze each symbol in context to its preceding ones so that they allow to detect anomalies localized within (long) sequences. They provide three different approaches to define the context of a symbol which reflect the different strengths of applying these techniques.

1.4 Hidden Markov Model-based Techniques

Hidden Markov Models [10] (HMM) are statistical models that provide powerful features for sequence modeling and, accordingly, for anomaly detection.

The key ingredient is a finite state machine with parameterized state transitions based upon a probability distribution reflecting the *Markovian property* (probability of a state transition depends on the previous state). Each symbol in a sequence is defined as an observation that leads to a state transition in the state machine which in turn delivers an output. The states and the state transitions constitute the *Markov process* which is not observable (therefore, the states are called *hidden states*). The output produced is the only observable indicator that provides the basis for anomaly detection e.g. in terms of labels or anomaly scores.

HMM-based anomaly detection techniques belong to the class of semi-supervised techniques. With respect to the available data, three different approaches may be applied to the learning process in the training phase([35]):

- for a given set of observation sequences, learn the most likely HMM parameters which result in maximum likelihood for the observation sequences
- for a given HMM, compute the hidden state sequence that is most likely to have generated a given test sequence

- for a given HMM, with given state transition and observation matrices, compute the probability of a test sequence

The key advantage of HMM it that they can model complex systems. However, the learning process may also be of high computational complexity. The efficiency of HMM applied to anomaly detection strongly depends on an adequate choice of (number of) state and parameters of state transitions in the initialization.

2. **Continuous subsequence-based anomaly detection**

Continuous subsequence-based anomaly detection is concerned with the detection of anomalous subsequences within a sequence. An anomalous subsequence is a significantly different subsequence within one sequence. The detection of continuous subsequence-based anomalies is a problem addressed by applications that produce monitored data over long periods of time. The basic technique applied to those problems is related to window-based anomaly detection techniques. The monitored data is split into fixed-size windows of length k. Each window is analyzed with respect to the rest of the windows in order to assign an anomaly score to it. Keogh et al. in [60] propose to compare those windows that do not overlap. This reduces the complexity and it intends decreasing mis-evaluations caused by similarities between overlapping windows.

Based on the basic technique, several variants exist addressing different aspects of the problem. First, different approaches exist for window scoring. While the simplest approach is based on counting the number of occurrences of a pattern within the database of windows, other approaches use distance measures related to nth nearest neighbor-based techniques (referring to Section 2.5.2 the performance of the anomaly detection is sensitive to the setting of n) or analyze windows by using compressions-based dissimilarity measures.

Comparing windows against the entire set of stored windows in the database is a very complex issue and usually requires $O(n^2)$ complexity. There are variants of the basic technique that aim to reduce the complexity and run approximately in linear time by immediately pruning such windows whose anomaly score is below a threshold (representing the pth largest anomaly scores).

A further aspect of the window-based problem is the to find an optimal value for the window size k. This is a challenging issue as a too small chosen k may lead to high false negative rates while a too large value for k may lead to high false positive rates. Considering that the length of anomalies is not known a priory and, furthermore, anomalies may be of varying lengths, setting the value of k becomes even more challenging. In their survey ([35]), Chandola et al. shortly discuss a segmentation-based approach that addresses the problem of finding an optimal k by exploring segments (or subsequences) of unequal lengths extracted from the sequence. However, acceptable solutions for this problem only exist for small alphabet sizes.

Techniques for sequence-based anomaly detection and continuous subsequence-based anomaly detection concentrate on different problem formulations and have been developed for different application scopes. However, with some adaptations that are discussed in [35], the techniques can be mutually applied to solve problems of the other type of problem formulation.

3. **Pattern frequency-based anomaly detection**

   Pattern frequency-based anomaly detection techniques deal with the problem whether for a given pattern, its frequency of occurrence in a test sequence is significantly different from its expected frequency. Thereby, the deviation in both directions (greater and smaller frequencies) is of interest.

   The basic technique tests for each query pattern the difference between its frequency in the test sequence with uts average frequency of occurrence in the entire data set. Based on this, the query pattern is assigned an anomaly score. Besides examining exact matching of query patterns with the database as performed by the basic technique, some variations exist that loosen the restriction of exact matching and allow some interspersion of symbols in the test pattern. For example, in one approach presented in [35], a test pattern is matched against the pattern data base in order to determine the longest substring in the test pattern that matches a pattern in the data base. This longest substring is used as a basis in order to count the frequency of that substring in the test sequence. Another approach referred by Chandola et al. is to count the occurrence of permutations of the query pattern in the test sequence if the ordering of the symbols is not significant.

   For counting the number of occurrences of a query pattern in a sequence, in particular, if the problem is not restricted to exact matching, the required computational complexity is huge. Chandola et al. refer to some techniques proposed by Keogh et al. that use so called *suffix trees* and which will be introduced in more detail in Chapter 4 (Related Work).

   Pattern-frequency based anomaly detection techniques can be applied to detect anomalous sequences and hence also address problems of sequence-based anomaly detection. In the contrary, these techniques cannot be applied to detect anomalous subsequences in a sequence, as they rely on a data base containing a set of sequences while the continuous subsequence-based anomaly detection approaches examine subsequences that occur in one long sequence.

For symbolic sequences, there are several techniques that have been proposed and developed within specific application domains and anomaly detection problem formulation. These techniques have been evaluated in their specific domains. Hence, they are not directly applicable to similar problems in other domains as it is not well-understood yet how a technique developed for one domain would perform in a completely different domain. Little deviations in the problem formulation, parameter configurations, type of available (training) data or the alphabet size may lead to totally different performance measures and efficiency of the applied technique.

2.5.9 Online Anomaly Detection Techniques

Many applications continuously generate or collect data during their operation. Some of them (e.g. critical applications) require analysis of the data online as soon as the data is available. Online analysis is the basis of online anomaly detection that enables immediate detection of occurring anomalies which in fact means real-time detection. Online anomaly detection, thereby, enables to induce immediate reaction on the anomaly in terms of preventive or corrective measures. The techniques discussed in the former subsections basically have been designed for offline (time-delayed) anomaly detection which, as a prerequisite, imply the complete test data to be available. In contrast to that, online anomaly detection by nature can only operate on the data that has been observed so far. From the techniques presented above, those techniques that unconditionally require the complete test data for analysis are not applicable to online anomaly detection (such as similarity-based techniques or pattern frequency-based techniques). However, some of the presented techniques can be either directly adopted or adapted to operate in online manner.

Continuously generated or collected data can be considered as sequence data so that if the data is of symbolic type, some of the techniques for sequence anomaly detection presented above can be exploited. In particular, window-based technique or subsequence-based techniques are suitable for online anomaly detection as they can assign anomaly scores to windows or even single symbols as soon as they occur. Furthermore, HMM-based techniques provide appropriate features to work in online manner as they rely on the fact that each observation is depending on its previous ones without taking into account those which may be upcoming.

Chandola et al. in [35] emphasize that all the referred techniques for online anomaly detection require an exhaustive training phase in order to become reliable in detecting anomalies.

2.6 Application Domains

Anomaly detection is applied in numerous application domains that use anomaly detection or do research in the field of anomaly detection. The specification of an anomaly and, hence, the objective of an anomaly detection approach is strongly related to the application domain and the associated purpose that it is implemented for. In this section, we discuss some typical applications for anomaly detection coming from different domains. As the community dealing with anomaly detection has been continuously growing in the recent years, the here listed and discussed domains are not exhaustive but provide an extract of the major application domains (surveyed by [34]). The most important application domain for the context of this thesis is covered by Intrusion Detection Systems (IDS), therefore it is separately introduced in an own section. The remaining application domains are summed up together in section 2.6.2.

2.6.1 Intrusion Detection

According to [34], Intrusion Detection plays an important role in computer security as it "refers to detection of malicious activity (break-ins, penetrations and other forms of computer abuse) in a computer related system...". From definition, intrusions exhibit different behaviors than the normal system behavior, and, referring to the definition of an anomaly, they can be seen as synonymous in this context. Therefore, anomaly detection techniques are applied to detect intrusive system behavior.

Anomaly detection in this context is based on system observation of events and logging data, but, as related to the nature of computing systems, huge amounts of such data is generated. Moreover, such data is generated in a more or less streaming manner. Therefore, the anomaly detection approach applied in this domain must be able to cope with analyzing huge data amounts and preferably, the applied approaches shall be able to process the data online. For computer systems, accurate and exhaustive training data is available as, usually, these systems can be run in testing and safe environments before being transferred into productive environments. The execution of the system in a test environments delivers activity data that is exploited for obtaining normal behavior profile. On the other hand, it is impractical to specify the behavior of all possible intrusions. Considering the evolution of intrusions, novel intrusions are continuously arising as intruders adapt the attacks in order to evade the present intrusion detection systems. However, in IDS the main objective is that the anomaly detection system is able to identify intrusions even if their behavior has not been known before as soon as it differs from the normal behavior profile. Therefore, referring to the classification of anomaly detection approaches in section 2.2, anomaly detection approaches applied to intrusion detection belong to the class of either *semi-supervised* or *unsupervised* techniques.

Intrusion Detection Systems (IDS) are classified into[27]:

- **Network IDS**: The aim of a Network Intrusion Detection Systems is to detect intrusions in network data related to a potential remote attack, e.g coming from the internet. Such intrusions can either be *Point Anomalies* initiated by one single event as well as *Collective Anomalies* formed by a sequence of events. Intrusion detection systems rely on data with different levels of granularity. One of the first developments in the domain of network intrusions systems and - most commonly referred - has been made by Anderson et al.[40] and is called NIDES.

- **Host-based IDS**: Host-based Intrusion Detection System usually reside in the operating system itself in order to protect the host from misuse. They are often referred toas system call based intrusion detection systems as the deal with operating system traces [34]. Single events in traces cannot be directly assigned to an intrusion as in terms of system calls for example, all system calls provided by the operating system are legal events in principle. But it is more the co-occurence of successive calls that may lead into a malicious system behavior. Therefore, anomalies occur mainly in form of *Collective Anomalies* within subsequences of traces produced by malicious programs, unauthorized behavior of policy validations. The traces may be produced by programs as well as users which each produce individual sequences of varying length and are typically sequential data items. Considering investigations to evade the system's intrusion detection, based on this customized profiles for each program or user, it is difficult for the attacker to know a normal behavior profile in order to adapt the attack in such a way that it converges to the according normal behavior.

  Many anomaly detection methods using different techniques have been developed in order to solve the host-based intrusion detection problem. As this application domain is most interesting regarding this thesis, existing approaches dealing with system call anomaly detection are examined in more detail in the Chapter 4.

2.6.2 Other Application Domains

Even if Intrusion Detection is the most interesting anomaly detection application domain, other application domains may face similar conditions and restrictions so that the may provide applicable solutions for the problem regarded here. The assumptions and conditions determined in an application domain can be used as guidelines to assess the effectiveness of the applied techniques for similar problems even though developed for this particular application domain. These are mainly presented in [34] (with no claim of providing a complete and comprehensive overview):

- **Fraud Detection**: Fraud Detection deals with identifying anomalies that represent criminal activities in commercial procedures such as bank, credit card, cell phone, insurance or stock market operations. Frauds are defined as uncommon consumer operations that may be caused by an unauthorized user. The objective of fraud detection algorithms is to immediately identify the misuse in order to prevent economic losses through *activity monitoring* (introduced by Fawcett and Provost in 1999 [42], referenced by [34]). For each user, an activity profile is maintained. Typically, such activity profiles are generated over several dimensions over periods of time and are continuously updated as the data collection is proceeded continuously whenever a transaction happens. This can lead to an expensive central storing of high volume of data caused by the nature of the application. On the basis of a users' activity profile, any new transaction is matched against the activity profile in order to detect any deviation online. In fact, the anomalies aimed to be detected belong to the class of *Point Anomalies* (considering an individual transaction being uncommon) and/or *Contextual Anomalies* (considering the context of the transaction being uncommon, e.g. geographical location as source of the transaction). As based on huge data amounts with numerous dimensions, this online detection may become complex. Typical solutions for fraud detection problems are provided by profiling and clustering-based techniques and belong to the class of either *semi-supervised* or *unsupervised* anomaly detection techniques.

- **Insurance and Health Care**: Applications in this domain have different perspectives and characteristics. For example, there are applications that aim to detect disease outbreaks on the basis of patients records which is a critical problem requiring a high degree of reliability and accuracy. Usually, anomalies detected by these applications are referred to *Point Anomalies* while the output of such an approach is of predictive qualities. Other applications concerned with anomalies in *Electrocardiograms* handle time series data in order to detect *Collective Anomalies*.

- **Fault Detection** (in safety critical systems or industrial/mechanical units): Anomaly detection in this context is applied to early detect or even predict failures of system units due to wearout circumstances in order to prevent damage of other system parts or actually the entire system. A normal profile of data is usually available based on the system specifications so that *semi-supervised* approaches are applied. Furthermore, the data of normal behavior is even of static manner.

  Monitoring is realized in an online fashion by sensors from which data that is gathered has temporal context. Anomalies in such systems can be determined by using time series analysis techniques and are *Contextual Anomalies* or *Collective Anomalies*.

- **Sensor Networks**: Sensors collect a lot of data that has to be analyzed in terms of interesting data items that differ from the rest. However, it depends on the application area of the sensor which kind of data it collects (binary, discrete, continuous data, etc.) even though the data is usually gathered in a streaming manner. Furthermore, the data is often required to be analyzed in an online manner. Deciding which techniques are applicable best is strongly related to the nature of the data as well as the types of anomalies to be detected.

- **Image Processing**: Anomaly Detection in image processing applications are intended to identify changes in images over time, also referred as motion detection, or finding changes in regions of static images. Chandola et al. enumerate satellite imagery, digit recognition, spectroscopy, mammographic image analysis and video surveillance as main application areas. Considering these applications, the big challenge for image anomaly detection is based on the large size of input data for analysis. As anomaly detection in this domain is interested in finding single points of interest as well as regions, anomalies are either of class *Point Anomalies* or *Contextual Anomalies*.

Chandola et al. have collected references to publications concerning all these application domains for further reading in [34]. They also list other domains that exploit techniques and approaches from anomaly detection such as text processing, speech recognition, novelty detection, traffic monitoring, fault detection in web applications, anomaly detection in biological data as well as astronomical data, detecting ecosystem disturbances, just to name some of them. In fact, the applicability of anomaly detection approaches is unlimited and particularly not restricted to these application domains.

2.7 Summary

This chapter provides an introduction into the topic of anomaly detection focussing, in particular, on the problem of developing an anomaly detection approach as addressed by this thesis. All challenging aspects concerning the development phase are pointed out. First, the development of an anomaly detection approach requires a clear formulation of the underlying problem according to the properties defined in this chapter. Then, design decisions that are summarized in Section 2.2.2 have to be made before the architectural model is mapped on the problem's application domain. As a next step, it is important to determine the classes applied to data classification. All theses decisions establish the basis to assess the applicability of specific anomaly detection techniques in terms of solving the problem addressed. Section 2.5 gives an overview on the best known anomaly detection techniques, their characteristics and application purposes, while Section 2.6 provides an overview over the common application domains for anomaly detection.

This chapter helps to classify the problem and the requirements on the anomaly detection problem examined in this thesis and emphasizes the challenges concerned with developing approaches for anomaly detection. Furthermore, with the information provided by this chapter, we can assess the potentials of existing techniques to solve our anomaly detection problem. From their limitations in terms of dealing with autonomous self-reconfiguring systems, we have derived the motivation to develop a better-applicable novel anomaly detection approach.

CHAPTER 3

Artificial Immune Systems

In this chapter, first Artificial Immune Systems are introduced in short including a discussion of their properties as well as their applicability to problem solving. This part of this chapter mainly refers to surveys provided by de Castro in [32] and [31]. Afterwards, an introduction is given to Danger Theory that builds up the core of the approach proposed in this thesis.

3.1 The Human Immune System

The human immune system is the defense system of the human body against diseases. Based on its powerfulness in detecting and protecting, and the processes taking place in the human body, it inspired the development of the computational paradigm of Artificial Immune Systems (AIS).

De Castro describes the immune system in [31, Chapter 6] as ”... an intricate collection of distributed cells, molecules and organs that altogether play an important role of maintaining a dynamical internal state of equilibrium in our bodies.” Its primary function is to protect the body from disease-causing agents such as viruses, bacteria and parasites, called *pathogens* by immunologists. It is capable of recognizing those *pathogens* and fighting against them by stimulating a so-called *immune response*.

The human immune system is a multilayer system composed of a physical barrier, which is the skin, a biochemical barrier in forms of pH level and saliva, the Innate Immune System and the Adaptive Immune System as illustrated by Fig. 3.1.

The Innate Immune System is understood as the first line of defense that regulates the Adaptive Immune System to induce the *immune response*. The cells belonging to the Innate Immune Systems are capable of recognizing a set of *pathogens* that possess generic molecular patterns. The immunity provided by the Innate Immune system is inherently given by birth.

The Adaptive Immune System primarily consists of white blood cells (called *B-cells* as maturating in the bone marrow and *T-cell* as maturating in the thymus) that are responsible for recognition of those molecular patterns that cannot be recognized by the Innate Immune System. These cells are able to adapt to previously unknown patterns through variation in genetic reproduction. When matching the unknown *pathogen*, these cells reproduce and thereby generate and maintain a stable memory of already known patterns. Having recognized a

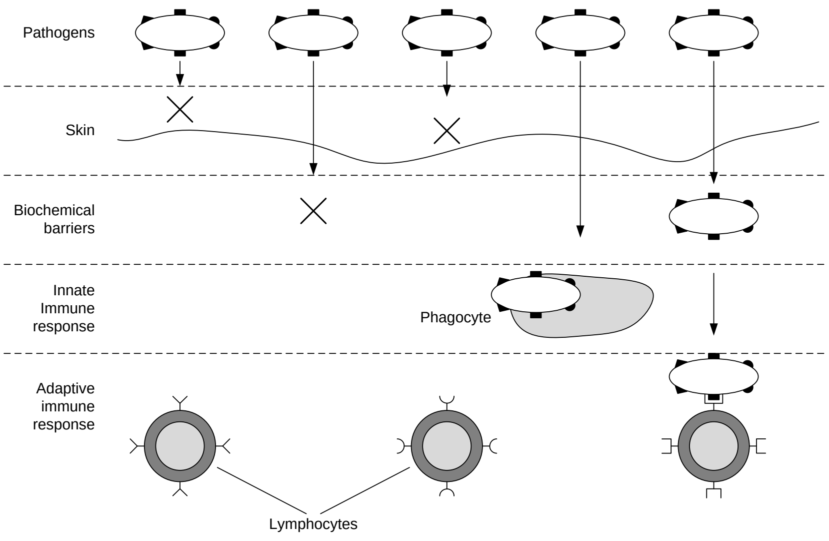


Figure 3.1: Multilayered Architecture of Human Immune System.

pathogen, the immune cells bind the *pathogen* and signal other immune cells of the Adaptive Immune System, such as big cell eaters known as *phagocytes*, to destroy the bound complex.

By this, the Human Immune System implements adaptation, learning, and memorizing. Different theories exist to explain the processes taking place in the immune system:

- Pattern Recognition
- Clonal Selection
- Immune Learning and Memory
- Affinity Maturation
- Self/Nonself Discrimination
- Immune Network Theory
- Danger Theory

We will not introduce these immunological theories here. For more insight into these theories, see [32]. However, we will present the computational principle derived from Danger Theory after giving an introduction into the computational paradigm of Self/Nonself Discrimination that to some extend builds the fundamentals of Danger Theory.

3.2 Artificial Immune Systems

Garrett emphasizes in [48] that there are two different understandings of Artificial Immune Systems reflecting two different viewpoints. On the one hand, referring to the computational immunology, ”an AIS is a model of the immune system that can be used by immunologists for explanation, experimentation and prediction activities” (see [48]) which in fact, emulates an immune system being implemented into a computing system. On the other hand, when considering the other viewpoint, ”an AIS is an abstraction of one or more immunological processes” (see [48]) and provides methods and algorithms that have been developed based on these abstractions in order to solve computational problems. The latter viewpoint is the one which we rely on in this thesis when referring to AIS.

De Castro provides a general definition for Artificial Immune Systems (see [32, Chapter 3]) that summarizes other definitions which can be found in the literature: ”*Artificial Immune Systems are adaptive systems, inspired by theoretical immunology and observed immune functions, principles and models, which are applied to problem solving.*”

From the immunological theories, computational models and algorithms have been derived (also providing abstractions of the immunological process) that altogether are described as Artificial Immune Systems. They are intended to be applied to solve complex problems by making use of the ideas, models and algorithms offered by AIS. The set of the features and characteristics provided by the models and algorithms of AIS is diverse and powerful (see [32, Chapter 3]): Pattern recognition, Uniqueness, Self identity, Diversity, Disposability, Autonomy, Multilayered, No secure layer, Anomaly detection, Dynamically changing coverage, Distributivity, Noise tolerance, Resilience, Fault tolerance, Robustness, Immune learning and memory, Predator-prey pattern of response, Self-organization, Integration with other systems, etc.

By constructing a defense system that is capable for fighting against unknown invaders in a robust and autonomous manner, the human immune systems forms a suitable model for computational anomaly detection. The most commonly known algorithms of AIS are:

- Clonal Selection Algorithm - CLONALG [39]
- Self/Nonself-Discrimination - Negative Selection Algorithm [47]
- Immune Networks Algorithms - aiNet [38]
- Dendritic Cell Algorithm (DCA) [52]

Typical application domains of AIS are [54]:

- Clustering/Classification
- Anomaly Detection (Intrusion Detection)
- Computer Security
- Optimization
- (Machine) Learning
- Bio-informatics
- Image Processing

- Robotics
- Control
- Virus Detection
- Data Mining / Web Mining

AIS are applied in Anomaly Detection as well as Learning, and therefore, it promotes to be applicable to the anomaly detection in self-x systems. Many publications exist that are concerned with discussing AIS, the models and algorithms, and its applications like [55], [37] and [21]. Furthermore, Chapter 4 provides a discussion on AIS-based approaches implemented for anomaly detection and refers to many well-known and often cited approaches.

3.3 Self/Nonself Discrimination

Self/Nonself Discrimination bases on the capability of the human immune system to differentiate between the system's own cells entitled as *self* and foreign cells depicted as *nonself*. This theory makes use of the process taking place in the thymus that is responsible for the maturation of T-cells. T-cells are intended to recognize *nonself* antigens (those pathogens that can promote an adaptive immune response). For this purpose, the immature T-cells are tested in the thymus against *self* antigens, which are the organism's self cells: If a T-cell is able to bind *self* antigens, it becomes useless for antigen detection. Furthermore, by providing receptors to bind organism's self cells, it potentially would provoke autoimmune responses. As this is not intended, those T-cells are immediately destroyed in the thymus. In contrast, T-cells that are not able to recognize and bind *self* antigens are expected to be capable of recognizing other cells not belonging to the organism, meaning the *nonself* antigens. These T-cells maturate in the thymus, become immune system cells that are released out of the thymus to circulate through the body in order to search for *nonself* antigens. This process is called *negative selection of T-cells* [31, Chapter 6] from which the most common algorithm arose: *Negative Selection Algorithm*.

This model forms the basis of anomaly detection in which *self* patterns are classified as *safe* while the *nonself* patterns are all patterns that do not belong to the system's known behavior and therefore endanger the system.

3.3.1 Negative Selection Algorithm

In 1994, Forrest et. al proposed in [47] the first algorithm inspired by the principle of *negative selection of T-cells* which then was named the *Negative Selection Algorithm*.

Like many other anomaly detection algorithms, its lifecycle is divided into a training phase and an operation phase. The objective of the training phase is to establish a set of detectors that are able to recognize *nonself* patterns. Patterns in this context can be binary patterns, real-valued patterns or any other patterns reflecting the behavior of a certain part of the system that is dedicated to be equipped with anomaly detection. The detector set generation process is illustrated in Fig. 3.2. For the generation of the detector set, the set of *self* patterns is required that is assumed to define the *normal behavior*. Detector strings are generated randomly and matched against the set of *self strings* by applying a predefined affinity function. If the detector string passes the matching process by matching at least one string of the set of *self strings*, it is able to recognize *self strings* and is therefore rejected by the pool of *nonself* detectors. In contrary, if for each *self string* the affinity function matching the detector string with the *self* *string* exceeds a predefined threshold so that the detector string is not able to bind *self*, it is declared as a *nonself* matching detector and therefore added to the detector set. This process is repeated until the set of detectors is big enough.

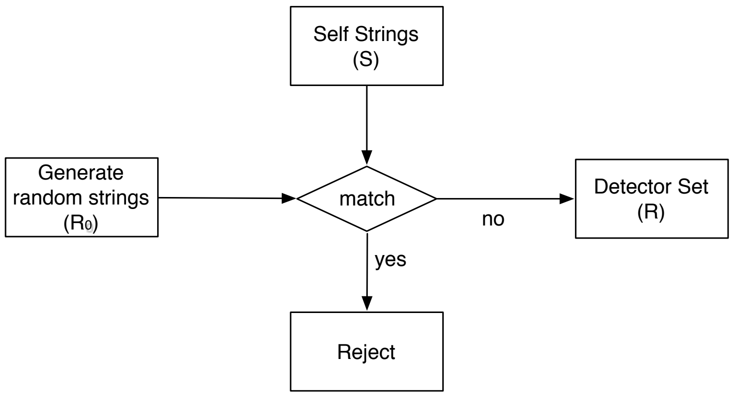


Figure 3.2: Detector Set generation in the Negative Selection Algorithm (source [31, Chapter 6]).

In the operational phase, any monitored pattern is then matched against the detector set. As long as the monitored patterns fail the matching test, they are classified as *self* patterns that represent *normal behavior*. If a monitored pattern matches at least one detector from the detector set, it identifies a *nonself* pattern that is classified as an anomaly.

The *Negative Selection Algorithm* was primarily developed for the purpose of computer security applications, such as virus detection or intrusion detection. But in the last two decades, it was migrated to other diverse application domains as well.

A analogical approach exists that is called *Positive Selection Algorithm*. This approach relies on the knowledge of *nonself* patterns that are used to train the detector set capable to recognize the *self* patterns when transferred into the operation phase. This approach is better applicable to systems where all *nonself* patterns are known in advance, and the set of *self* patterns has much higher variability, and is much greater than the set of *nonself* pattern.

3.3.2 Limitations of Applicability to Self-x Systems

The *Negative Selection Algorithm* offers potentials to anomaly detection as it is able to differentiate between *normal* behavior and such behavior that deviates from *normal*. However, the strength of this approach relies on the definition of *normal* that is specified in the set of *self strings* as these are exploit to build up the detector set in the training phase.

For autonomous systems that exhibit dynamic behavior, a definition of *normal* behavior is very challenging, if not impossible. Even if it would be possible to specify the *self* strings for one system configuration, with a high probability they might become invalid after reconfiguring the system. The system is immediately continuing its execution after a reconfiguration so that there is no time for the anomaly detection module to switch into a training phase. Last but not least, the reconfiguration is performed autonomously leading to a configuration that is not predictable. If applying this approach in the context of autonomous systems, the questions occur which training set to use and whether it actually is possible to generate training sets of all possible configurations - as in the motivation of the present work, we have already stated that not all system behaviors can be foreseen at design time.

Furthermore, at operation phase, another problem is faced: with the requirement to work in an online manner, the targeted approach is restricted in terms of computation time as applied in a real-time system. Hence, if the detector set is huge (as it has to cover all possible *normal* behaviors of the dynamic system) in order to ensure an adequate coverage, it becomes impractical to match an actually monitored behavior against the complete set of detectors online as this would introduce enormous execution complexity into the system.

Based on these arguments, *Negative Selection Algorithm* was assessed to be not applicable for the purpose of this thesis. However, its principles build up the basis for the *Danger Theory*. For better understanding the motivation of and the differentiation to the Danger Theory, a short introduction is noteworthy here.

3.4 Danger Theory

The Danger Theory was proposed by Matzinger in 1994 and published in the Annual Review on Immunology [76]. It was motivated by the fact that Self/Nonself Discrimination failed in explaining some phenomena observed in the human immune system, e.g.

- foreign or *nonself* cells are tolerated and not destroyed, such as in case of pregnancy
- foreign or *nonself* cells should not be destroyed in cases of transplantations
- own or *self* cells can also endanger the system, when they mutate to cancer cells

The main questions derived from this are: what happens when *self* changes? what if *self* becomes harmful? how can harmless *nonself* be tolerated? etc. These questions are closely linked to the challenges of the targeted anomaly detection approach.

One of the main challenges related to these phenomena is to explain how the immune response is triggered as only differentiating between *self* and *nonself* turned out to be insufficient. From the immunological point of view, the basic idea of the Danger Theory refers to the fact that cells undergo injury, stress or bad cell death - that is termed *necrotis* -, and thereby they send out *danger signals*. Antigen-presenting-cells (APC) are sending activation signals based on the dynamic and constantly updated presence of *danger signals* that trigger the immune response (accomplished by T-cells). Detailed exploration of the theory can be found in [76]. Until today, the theory is treated controversial among immunologist as it could not be proven completely (see [66]). Nevertheless, it provides a powerful computational model that can be exploited for solving computational problems such as those in the domain of anomaly detection.

3.4.1 Computational Danger Theory

Like the other immune system principles, the Danger Theory provides a good metaphor and a promising model for computational purposes offering powerful properties and features. Therefore, computer scientists like Aickelin et al. [20] propose a computational model of Danger Theory contributing to Artificial Immune Systems without any relevance of its actual existence in the human immune system.

Aickelin et al. [20] point out that the Danger Theory is relevant for approaches for which a discrimination of *self* and *nonself* is unsuitable in cases if *nonself* is not inherently associated with danger, if *self* is inaccurate or if the boundary between *self* and *nonself* may be blurred. Furthermore, the Danger Theory is applicable on problem domains where *self* is expected to change over time. The main contribution of the Danger Theory is that the system responds to the presence of danger signals.

Aickelin et al. [20] formulate aspects that have to be considered when incorporating the Danger Theory into a computational approach:

1. an entity responsible for presenting an appropriate danger signal as a counterpart for the antigen-presenting cell
2. the semantics of the danger signal (the danger signal may have nothing to do with danger, but can signal a *case of interest*) and its representation (positive/negative, presence/absence, single-valued/array, etc)
3. the definition of a danger zone: entities effected by a danger signal, temporal enduring effect of the danger signal, etc.
4. the immune response (that should not induce a danger signal itself)

Considered as an extension of Self/Nonself Discrimination, the Danger Theory allows to classify patterns as:

- self - harmless
- self - but harmful
- nonself - harmful
- nonself - but harmless

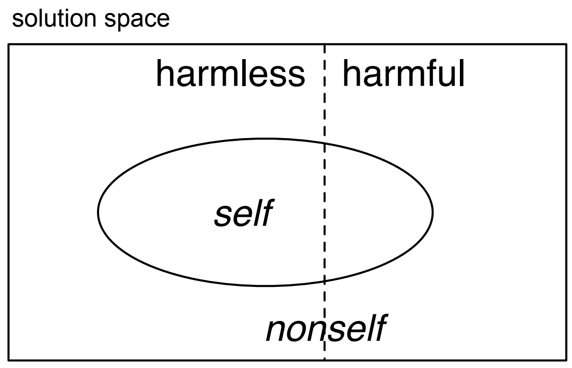


Figure 3.3: Classification of entities applying Self/Nonself Discrimination with Danger Theory.

Fig. 3.3 illustrates the different classification classes: the entire solution space consists of a set of *self* and its complement, the set of *nonself*, and is split into the classes harmless and harmful. The set of *self* can belong to the harmless class but partly also to the harmful class. Its complement, the set of *nonself* can also be part of harmful as well as harmless. With this classification, the immune response is triggered by a co-stimulated method as it is dependent on the second signal (the danger signal). Furthermore, the Danger Theory with quite a number of elements (see [76], [20]) provides potentials to alter the model according to the addressed problem. For example, it is possible to completely omit a discrimination into *self* and *nonself*, and operate only on the danger signal caused by distressed system entities.

Aickelin et al. [20] propose a number of application domains for the Danger Theory such as computer security, (hardware) fault detection, virus detection, intrusion detection but also anomaly detection in general, data mining and classification. Up to now, numerous publications exist that try to survey the most interesting applications which are making use of the Danger Theory.

3.4.2 Dendritic Cell Algorithm

One of the main contributions to Artificial Immune Systems from the Danger Theory is the Dendritic Cell Algorithm. Dendritic Cells (DCs) are white blood cells that are assumed to act according to the model proposed by the Danger Theory (their functionality is not fully examined yet). Hence, the Dendritic Cell Algorithm was derived from the properties of DCs and was developed by Greensmith, Aickelin and Cayzer [51] for the purpose of anomaly detection.

Immunological Background of Dendritic Cells

From an immunological point of view, Dendritic Cells are antigen-presenting cells that are responsible for the major control mechanism of the immune system as acting as a vital interface between innate immune system, performing the initial recognition, and the adaptive one executing the effector response. Basically, the DC's function is to collect antigens from pathogens or the host cells in tissue, and present these to the T-cells in the lymph nodes. Besides the initial recognition, DCs sense the environment and differentiate behavior based on the concentrations of the present signals.

For a DC, three states are defined: *immature*, *semi-mature* and *mature*. The state of the DC is determined by the environmental signals or events, or the correlation of various signals. The *immature* state is the initial state of the DC. In this state, it collects and processes material of its environment. The *immature* DC can maturate either into the *mature* or the *semi-mature* state. The maturation process is controlled by the differentiation of the collected material in respect to the receiving signals. If the *immature* DC is exposed to a certain number of (threat or danger) signals, it maturates its surface in order to become able to stimulate T-cells and thereby becomes a *mature* DC. In absence of those signals, or in case of a too small concentration of signals, the *immature* DC changes into *semi-mature* state. The *mature* DC has the ability to activate naive T-cells that induce an immune response, while in contrary, the **semi-mature** DC has a suppressive effect on the immune response. After maturation, a DC, irrespective of whether in the *mature* or the *semi-mature* state, migrates to the lymph node where it is conserved.

The Computational Counterpart

From the desirable characteristics exhibited by DCs, Greensmith et al. [51] have derived an abstract model of the DC interactions and functions. The core properties describing this model are:

- the initial state of the DC is the *immature* state from which it transitions to either *mature* or *semi-mature* state
- the immature DC collects information it is dedicated for (each DC can have assigned an individual purpose)

- maturation is caused by the exposure of certain signals received by the DC
- the output of DCs after maturation is different for mature DCs and semi-mature DCs and provides contextual information

DCs are treated as processors of signals that costimulate the classification of the collected data. Greensmith et al. [51] propose the following input signals:

- PAMP (pathogen-associated molecular pattern) - exposes pattern of a known threat
- Danger Signal - exposes a potential danger
- Safe Signal - represents a safe state
- Inflammatory Signal - general alarm signal (positive or negative)

These signals are present (with different concentrations) in the environment of each DC. From a biological viewpoint, PAMPs are molecules that are exclusively expressed by invaders (pathogens such as bacteria, viruses, etc.) that are inherently infectious *nonself*. The recognition of these molecules identifies a threat. According to the biological counterpart, the PAMP signal is defined as a known danger. In the computational context, the PAMP signal can be anything that definitely identifies a thread, such as a known dangerous behavior pattern or signature or an attribute value exceeding a predefined threshold (e.g. number of error messages). PAMP signals can either be specified by expert knowledge at the specification phase and/or they can be learned at operation phase based on behavior patterns that previously have been detected as leading to a danger and have been memorized. Danger signals are representing potential threats based on internal (state) information that indicate high level of anomaly, such as receiving a high number of network packages per second. Both, the PAMP signal and the danger signal induce the DC to migrate into the mature state. However, the PAMP signal entails the most intensive signal for the DC to become a mature DC while the danger signal offers tendencies of the DC to migrate to the mature state in case of a high concentration of the signal.

In contrast, the safe signal shows up in a safe state of the DC's environment without potentials of threats and, hence, it promotes the DC to migrate to the semi-mature state. This maturation process is illustrated by Fig. 3.4.

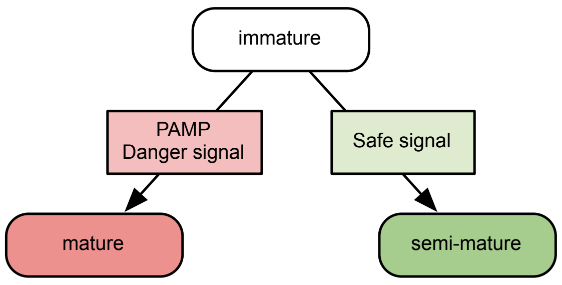


Figure 3.4: Maturation of Dendritic Cells.

The Inflammatory Signal is not sufficient to initiate maturation of a DC, but it can be considered as a generous alert signal that amplifies the concentration of the remaining signals in order to enhance the maturation tendency.

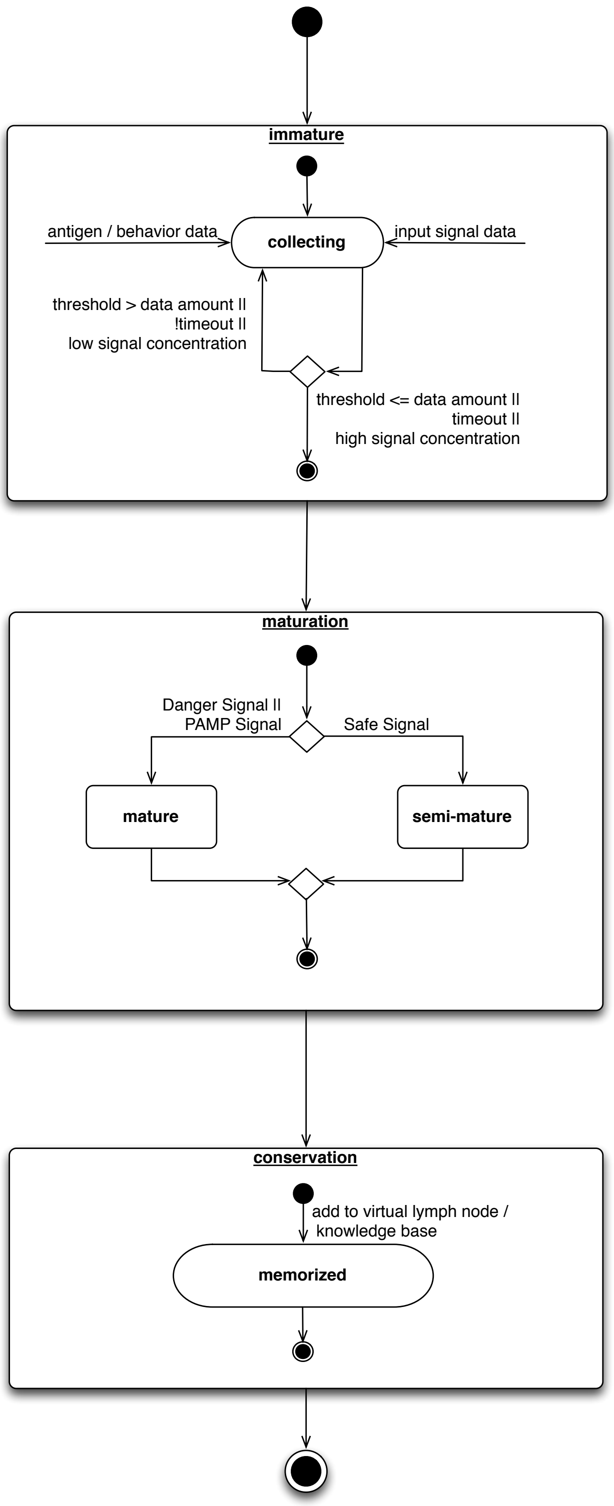


Figure 3.5: Lifecycle of a Dendritic Cell.

The lifecyle of a DC is illustrated in Fig. 3.5. At the beginning of a DC's lifecycle, the DC is in the initial immature state collecting environmental data. The DC will reside in that state until it will either reach a predefined amount of collected data specified by a threshold, or until a specific condition for state transition is present e.g. a combination of the concentration of costimulatory input signals or a timeout. Then, the DC performs the maturation by migrating into either the mature or the semi-mature state. The migration is governed by the concentration of danger signal, PAMP signal and safe signal. After the state transition, the DC will go into the conservation state to enter a virtual lymph node with its maturity context for the purpose of being memorized.

According to its biological counterpart, the computational approach is a population-based approach involving multiple DCs having assigned individual tasks (of course overlapping with other DCs' tasks). Population-based approaches are well known for ensuring robustness as they are eliminating single-point-of-failures and in the context of anomaly detection they can reduce the risk of false alarms.

Greensmith et al. have shown the functionality and the efficiency of this algorithm in [51] using the example of detecting anomalies in breast cancer data-sets. Furthermore, in [52], Greensmith et al. provide a detailed description of the Dendritic Cell Algorithm including also pseudo code.

3.4.3 Applicability to Anomaly Detection in Self-x Systems

In contrast to Self/Nonself Discrimination, the Danger Theory does not rely on a separated training phase to establish a *normal* behavior knowledge entitled as *self*. When applying the Danger Theory, the collected behavior data is not matched against any previously trained knowledge but it is rather assessed in the context of the observed environmental signals. Hence, changes in *normal* behavior do not effect the effectiveness of approaches applying the Danger Theory as the currently collected behavior data is not compared to any (history or trained) behavior data. The immune system implemented in forms of the Danger Theory is only dedicated to induce an immune response in case of present threats that are represented by correlations of various signals. This characteristic, in particular, is essential for self-reconfiguring systems in which the *normal* behavior may vary and yield in behavior previously unknown. These previously unknown patterns do not need automatically to be classified as threats but, in contrast, they are examined with respect to their contextual environment signals which makes this approach applicable to anomaly detection in autonomous systems.

Furthermore, the Danger Theory does not require huge amounts of stored data to assess the currently collected data. Contrary, it operates only on the data present at the specific time stamp and, therefore, becomes applicable to such resource restricted systems like those that we address in this thesis.

The input signals that are used for defining the context of the collected data and, thereby ,allow to classify the currently collected data. This is well suited to the requirements formulated in advance. The PAMP signal indicating already known threats can be exploited properly exactly for that purpose: system parameters that definitely identify malicious system states are PAMP signals. The same applies to the danger signal that indicates a suspicious system state showing up a potential threat, and the safe signal reflecting a safe system state, accordingly.

The Danger Theory provides suitable characteristics and properties to be applied in the anomaly detection for self-x real-time systems and, therefore, it essentially inspired the presented work to develop a context-related classification of system and application behavior. Of course, applying Danger Theory requires to define the computational counterparts for the elements building up the Danger Theory, meaning to determine the data that is collected by the DCs, the components responsible for delivering the costimulatory signals, and finally, the virtual lymph node to conserve the DCs with their maturity information.

3.5 Evaluation of Artificial Immune Systems

Usually, the evaluation of Artificial Immune Systems is not straightforward. Simon Garrett investigates the questions concerned with how to evaluate Artificial Immune Systems in [48]. The main question Garrett discusses is whether and to what extent these novel approaches provided by AIS are *useful* and in his discussion he considers the individual paradigms of AIS (Self/Nonself Discrimination, Danger Theory, Clonal Selection and Immune Network Theory) in separate. In this context, Garrett defines the term *useful* as being a combination of *distinctiveness* and *effectiveness*, for which he also delivers definitions and explanations:

1. **Distinctiveness**:

   Distinctiveness deals with of the issue how distinct AIS paradigms are from other approaches. This issue is mainly related to the problem whether a paradigm provided by AIS can be transferred to any other paradigm originating from other sources of inspiration resulting in the same mathematical method or algorithm. In order to make distinctiveness of a novel method clearer, Garrett provides three main questions (Source [48]):

   D.1 Does the new method contain new symbols (and features)?

   D.2 Are the new method's symbols organized in novel expressions?

   D.3 Does the new method contain unique processes?

   The more of these questions are true for the novel approach, the more likely the approach can be classified as being *distinctive*, especially if it can be said that the symbols, expressions, and processes of the new approach as a whole cannot be made equivalent to other existing approaches.

2. **Effectiveness**

   Effectiveness deals with the issue of an approach of standing out in terms of being better or faster that in others. For assessing effectiveness, Garrett also provides three main questions to be considered (Source [48]):

   E.1 Does the method provide a unique means of obtaining a set of results?

   E.2 Does the method provide better results than existing methods?

   E.3 Does the method allow for a set of results to be obtained more quickly than another method?

   Concerning the declaration of an approach being *effective*, it is sufficient if only one of these questions is valid for a subset of problems that are covered by the evaluated approach.

As there might be examples for approaches being distinctive but ineffective and others being effective but not distinctive (reducible to other paradigms or algorithms), Garrett emphasized that *usefulness* of an approach presumes the coexisting presence of *distinctiveness* and *effectiveness*.

For Self/Nonself Discrimination (or Negative and Positive Selection), Garrett refers to Hofmeyr and Forrest [57] who point out that the process of learning rules introduced by Negative Selection is *distinctive* compared to other classifier learning methods. Considering the effectiveness of Negative Selection, it offers unique potentials to detecting different sets of objects by the variety of learned detectors and, in some cases, it exhibits more precise results in change detection (see [48] for further references), and hence it can be defined as *effective*. Due to these arguments, the AIS paradigm of Self/Nonself Discrimination, in fact, is assessed to be a *useful* approach as it is *distinctive* as well as *effective*.

Garrett does not offer any sophisticated validation of Danger Theory based on the presented evaluation method, as research on Danger Theory has been in its infancy when Garrett published his article in 2005. But as Danger Theory introduces novel processes, he acknowledges that it will obtain relevance in main areas as it promises the ability to find solutions for different problems (such as intrusion detection) in an autonomous manner.

3.6 Summary

This chapter gives an introduction into Artificial Immune Systems including a background on the source of inspiration for it, the Human Immune Systems. Their characteristics and properties as well as typical application domains are described. As one of the main principles, the approach of Self/Nonself Discrimination is explained in more details as its limitations built up the motivation for specifying the Danger Theory, the core inspiration of this thesis.

The core of the Danger Theory relies on Dendritic Cells and the corresponding computational approach that was introduced by Greensmith et al. [51]. Dendritic Cells operate on means of costimulatory input signals to classify collected data without any matching against *self* or *nonself* knowledge. This property provides the most essential concept to be applied for anomaly detection in self-x systems.

In this chapter, we discuss the applicability of the Danger Theory in the context of this thesis. Finally, we provide a discussion based on the publication of Garret [48] on the evaluation of Artificial Immune Systems.

CHAPTER 4

Related Work

In this chapter, we present approaches that, on the one hand, initially inspired the idea of anomaly detection developed in this thesis. On the other hand, limitations of these approaches and their nonapplicability in terms of addressing the challenges and requirements defined for an anomaly detection for self-x real-time systems are worked out to emphasize the necessity for a novel approach like the one presented in this work.

The objective of the targeted anomaly detection is to ensure the stability of the system on the basis of analyzing the system behavior. System behavior is mainly governed by the executing applications which through dedicated interfaces, usually in forms of system calls, interact with the operating system and use services of the operating system. Only by executing system calls, applications can effect the operating system's internal state. Thereby, particular sequences of system calls may offer potential to do harm. Hence, the system behavior can be substituted by the application behavior composed of system calls that become the observable objects for anomaly detection.

Analyzing system call sequences in order to detect anomalous behavior that may yield into threats for the concerned system was primarily brought in by Forrest et. al in [56]. Since then, many researchers have investigated into concepts and models for system call-based anomaly detection. These approaches are often dedicated to the closely related domain of host-based intrusion detection systems, predominantly addressing ordinary computer systems (non real-time). Nevertheless, they provide interesting aspects in terms of assessing what is applicable to our problem so that the most relevant ones will be discussed in short in Section 4.1.

By [47, 57, 46], and [44], Forrest and Hofmeyr (et al.) mainly pushed on the research in system call-based anomaly detection by applying concepts of Artificial Immune Systems. The concepts developed by Forrest and her co-researchers but also those of other researchers using the idea of Artificial Immune Systems (AIS) are discussed in Section 4.2.

Finally, as the Danger Theory is a specific principle belonging to Artificial Immune Systems, Danger Theory-inspired approaches for anomaly detection are presented in the last part of this chapter, in Section 4.3.

4.1 System Call-based Anomaly Detection

Approaches that examine system call sequences for detecting anomalies rely on the assumption that a particular class of system threats will be unveiled in unusual sequences of system calls [46].

As a first approach in this context, in [46], Forrest, Hofmeyr et al. have proposed an approach for look-ahead pairs of system calls for UNIX processes. They have monitored the execution traces of UNIX programs such as `sendmail` or `lpr` and derived from their traces short sequences of k system calls to build up a database of *normal* behavior by recording for each system call all the system calls that follow this call at the positions 1 up to $k-1$ in the sequence. Forrest, Hofmeyr et al. claim that if program code is static, a stable database can be obtained even by short sequences of length $k = 6$, as they have shown in [46]. By using the look-ahead pairs, this approach allows to take into account orderings of system calls within long system call traces. The look-ahead pairs of system calls are stored in tables, one for each process, that are used for matching newly occurring system call sequences against the *normal* behavior database. The purpose of the matching procedure is, first, to verify their presence of absence of the sequence in the database and, second, count the pairwise mismatches if the sequence is not present in the database. The number of mismatches dictates whether a currently occurring system call sequence is classified as *normal* or *anomalous*. The matching procedure can run in $O(N)$, where N is the length of the system call trace, and is therefore efficient to be executed online.

Even though this approach is stated as an initial step for building up an Artificial Immune System as the idea is based on the immunological principle of distinguishing between self and nonself by previously generating a *normal* behavior database, it does not implement any further immunological principles. Therefore, from my viewpoint, it does not belong to the class of AIS-inspired approaches. Nevertheless, this approach is one of those that inspired this thesis as it demonstrates simplicity in terms of detecting anomalous system call sequences with low runtime effort. However, the authors emphasize that the size of the database is strongly depending on the variations in the system call sequences and the length of the trace that is considered to build up the *normal* behavior database. The database may become huge in case of strong variations. Furthermore, it is a challenge to decide how much data is sufficient to obtain a stable *normal* behavior database in order to maintain runtime efficiency, on the one hand, as well as to achieve a good coverage on the other hand that is mainly responsible for preventing false classification results. Last but not least, it is the question how to set the threshold that determines a sequences to be either *normal* or *anomalous*. In particular, in the context of self-x systems, learning of changes in *normal* behavior are not taken into account by this approach which mainly made this approach not applicable for our purpose.

With this work ([46]), Forrest, Hofmeyr et al. started investigating into the topic of system call-based anomaly detection. Beyond this, they have refined their approach for the purpose of being applied for UNIX systems and examined a number of other mechanisms to distinguish between *normal* or *anomalous* behavior. One of these approaches was published in [56]:

By [56], Forrest, Hofmeyr et al. are actually cited by many other researches to have been the first ones who have proposed a system call-based anomaly detection (for intrusion detection in UNIX systems). This approach relies on measuring the distance (Hamming distance) of a fixed length system call trace executed by a program against its *normal* behavior database. If the distance value exceeds a specified threshold, the observed behavior sequence is classified as an anomaly or intrusion (as Forrest, Hofmeyr et al. prefer to use this term). In their experiments, Forrest, Hofmeyr et al. found out that not only for each program an individual *normal* behavior database is generated, but furthermore, that the *normal* behavior of a program differs with respect to a particular system (with a specific configuration) by which the diversity of this approach could be demonstrated.

In this approach, each program is assigned a *normal* behavior database that is established in a training phase (either in a synthetic or a real environment) by monitoring the system call execution traces for that program and storing k-length sequences of the system calls extracted from the trace in a rooted tree structure. Because UNIX programs are usually running continuously, Forrest, Hofmeyr et al. have chosen to use sliding windows in order to obtain all system call sequences of length k across the entire execution trace. By this, the database holds all k-length sequences having occurred during the training phase, which may become large for programs that use a large number of system calls. Forrest, Hofmeyr et al.[56] have build up such a database exemplarily for the UNIX program `sendmail` that uses 53 of the system calls provided by UNIX. Based on empirical evaluations, Forrest, Hofmeyr et al. have determined $k = 10$ as a good choice. With these parameters, the *normal* behavior database for `sendmail` includes 1318 unique system call sequences.

In the operation phase, any k-length system call sequence is matched against the database for the purpose of identifying whether the sequence is belonging to the *normal* behavior. This can be performed in constant time based on the fixed length of the sequence. However, if the sequence is not part of the *normal* behavior, the Hamming distance has to be calculated. Basically, this can only be conducted by pairwise comparison (the current sequence with each of the sequences stored in the *normal* behavior database). The complexity of this procedure is depending on the size of the *normal* behavior database. For large databases, this procedure becomes very time consuming (see the example of 1318 database entries for `sendmail`). As Forrest, Hofmeyr et al. extract the sequences by sliding windows, each new system call generates a new sequence that requires the execution of the distance calculated in worst case (in case the sequence is not part of the *normal* behavior database). In their conclusion, Forrest, Hofmeyr et al. emphasize that detecting anomalies or intrusions online requires careful attention to efficiency issues that they did not consider at the point of time their work was published. The results achieved in anomaly detection by this method have been better than those in [46], as more accurate detection precision was possible. However, from the viewpoint of the requirements set by real-time systems, matching each new sequence against the entire set of sequences in the *normal* behavior database becomes unfeasible due to timing constraints.

Besides the challenge of determining a suitable value for the variable k, choosing an adequate value for the threshold that causes a sequence to be either classified as normal or anomalous in another problem. From my viewpoint, there might be system call sequences i that differ only by one system call from a *normal* sequence, having a Hamming distance of $d(i) = min\{d(i,j)\ \forall$ normal sequences $j\} = 1$, that could already endanger the system. On the other hand, there might be sequences with a greater value for $d(i)$ that would not. Forrest, Hofmeyr et al. have not formally or theoretically proven how to generally define an adequate threshold value but, moreover, they have set this threshold based on empirical evaluations that have to be individually performed for each particular program.

Another more important aspect which makes this approach not applicable in the context of the problem addressed by this thesis is that the *normal* behavior database is established during a training phase. Thereby, the knowledge base becomes fixed for the entire system execution which is suitable for standard computer systems. In environments with self-x capabilities, behaviors of applications can change dynamically so that *normal* behavior may alter but become uncovered by the *normal* behavior database. It is not possible to predict any possible configuration that may result from autonomous decisions in order to provide a *normal* behavior database for each configuration. Furthermore, it is impractical to induce a training phase after each system reconfiguration while the system is in execution as due to autonomous decisions false reconfiguration decisions may cause behaviors that may immediately endanger the system.

Numerous publications on system call based anomaly detection exist that offer good surveys on the research taking place in this domain. E.g. in [93], Warrender, Forrest et al. provide a first discussion of models for system call data. Chandola et al. [36] provide a comparative evaluation of anomaly detection techniques for sequence data by which they mainly address system call sequences. In a later publication in 2008, Forrest, Hofmeyr et al. [44] have summarized research taken place in system call-monitoring and its application to anomaly intrusion detection since their first paper in that topic. In [93] and [44], the authors have mainly set focus on analyzing the diverse data models for representing *normal* behavior applied by various researchers they reference. Apart from their own concepts concerning look-ahead pairs of system calls, tree-structure representation of *k*-length sequences, etc., they refer to approaches that model *normal* behavior by statistical analysis (based on frequencies), Finite State Automata, Markov Models, Hidden Markov Models, Neural Networks, Bayes models, fixed-sized windows as well as windows-based approaches with variable size, etc. In addition to that, Chandola et al. [36] enumerate Probabilistic Suffix Trees as a further data representation model. Forrest, Hofmeyr et al. state that "developing a normal profile is a typical machine learning problem" [44]. Each of the approaches applied are aimed at producing a more accurate representation of *normal* - irrespective if in a supervised or unsupervised learning manner - in order to obtain a stable *normal* that enables to reduce false rates.

Forrest, Hofmeyr et al. emphasize that for an anomaly detection capable to work online it is essential for the representation of *normal* to remain compact. Not all of the referenced approaches meet this requirement. Furthermore, in association with the data model, different algorithms and methods are applied to differentiate between *normal* and *anomalous* with different degrees of precision and different computational complexity. Beyond that, some approaches such as [72] additionally take into account system call arguments and parameters in order to detect inconsistencies which introduced much more complexity into the definition of *normal* as well as into the classification method. Through the diversity of the existing approaches, a comparison between them is difficult or even impossible. Each of the approaches outperforms others in a specific application purpose, specific system configuration or environment. However, approaches with a complex structure for representing *normal* or those using methods with high computational complexity for evaluation of executing sequences - even if they achieve a dedicated accuracy - are inapplicable for executing online as they may enormously interfere the system execution. In the context of real-time systems, such additional computational overload may lead to infeasible system configurations that could violate the system's as well as the task's real-time constraints.

A more critical deficiency pointed out by Forrest, Hofmeyr et al. is that all these approaches are not able to cope with dynamically changing behavior where *normal* behavior may evolve over time. Once, *normal* is established in a training phase either through synthetic generation or extracted by real execution, it is fixed with no mechanisms incorporated for adaptation or adjustment of *normal* during system execution.

4.2 AIS-inspired Anomaly Detection

Forrest et al. [45] state that there are compelling similarities between the problems of immune systems and computer system. Therefore, they propose to apply principles found in immunology to computer science problems. Especially for defense purposes in computer systems AIS are assumed to be promising and powerful because of their effectiveness in defense of the natural counterpart, the Human Immune Systems (HIS). They provide a valuable source of inspiration for anomaly detection because of their ability to protect against previously unknown invaders in dynamic environments with increasing complexity in an autonomous and adaptive manner (see Chapter 3). Furthermore, principles derived from the HIS and formulated in computational approaches that contribute to AIS are characterized as lightweight and robust so that they are expected to be better applicable in online manner than the traditional approaches mentioned in the previous section.

The first approach known to us that can be called immune system-inspired was published by Forrest et al. in [47]. This approach is dealing with self/nonself discrimination for differentiation between legitimate and harmful activity in computer systems for ensuring their security. The work published by Forrest et al. [47] aims to develop a change detection method for identifying changes caused by viruses or unauthorized users that was inspired by AIS. In particular, Forrest et al. apply the principle of self/nonself discrimination where *self* is represented by (equally-sized) strings reflecting legitimate behavior. Even if Forrest et al. did not address the problem of system call sequences directly, system call sequences can be described by strings consisting of the IDs of system calls ordered according to their temporal occurrence which makes the approach also applicable to system calls, or sequences respectively. For enabling the detection of deviations from *self*, the generation of random detectors is performed in a censoring phase (that we call training phase). The detector generation was inspired by the generation of T-cells that was later formulated as the *Negative Selection Algorithm* (see Chapter 3.3.1). A set of unique detectors is established of which each is able to bind different kinds of *nonself* strings founding distributivity and diversity of that approach. Different matching functions were proposed to be applied such as exact matching, or matching of r-contiguous symbols of strings. The authors emphasize that the generation of the detector set is computationally expensive, but checking for a deviation from *self* is computationally cheap. By demonstrating the simplicity and the strength of their AIS-based approach, Forrest and her co-researchers have become forerunners in the domain of anomaly detection and have predominantly contributed to the raising interest in employing AIS for anomaly detection.

Based on the approach proposed by Forrest et al., widespread research has been conducted in the field of employing immune systems in anomaly detection, in particular to be integrated into anomaly-based Intrusion Detection Systems (IDS). Aickelin, Greensmith et al. provided a detailed review of the existing research approaches in [61].

First, they summarize works exploiting self/nonself discrimination as a principle. This class of approaches defines *self* and builds up a *normal* database by making use of conventional algorithms. Also, classifying the activity monitored at the system execution is performed by traditional classification methods. For these approaches, only the idea of discrimination between *self* and *nonself* is exploited so that the immune system is set as an on top architecture, mimicking the HIS at a high abstraction level. Hence, the performance and efficiency of those approaches is only related to the characteristics of the algorithms and methods applied.

In their review, a main focus is set by Aickelin, Greensmith et al. [61] on research applying the Negative Selection Algorithm for anomaly detection in IDS. The main motivation of all the approaches applying Negative Selection resides on the ability to identify *nonself* - and thereby meaning invaders or intrusions - without prior knowledge of their structure or appearance as it is found in the HIS. Aickelin, Greensmith et al. [61] give an exhaustive review of the existing approaches including their specific characteristics, their performance and their uniqueness in relation to others. Therefore, we are not interested in repeatedly summarizing the main aspects of the discussed approaches and rephrasing the analysis made by the authors (for further insight into the approaches referred to, see [61]). Moreover, here, we want to point out general commonalities, differences, powerfulness and limitations of this class of approaches:

By applying Negative Selection, all these approaches enumerated by Aickelin, Greensmith et al. have in common the proceeding of the following three phases:

1. Defining *self* and an adequate representation of *self* patterns
2. Generation of detector sets
3. Classification of monitored activity on the basis of the detector set

Apart from this common procedure, the existing approaches differ in their realization of these three phases. There are different points of interest in IDS solutions to determine *self*. On the one hand, *self* can be defined as internal system data of an operating system component in case of host-based IDS. One option is to define system call sequences executed by applications as *self*, in which we are mostly interested in this thesis. Furthermore, as IDS are applied in computer systems that are interconnected, also network connections are considered as point of interest to be monitored if incorporated into network-based IDS. Then, *self* is defined according to that purpose. The representations of *self* are also varying with respect to the data that is determined to be meaningful showing up characteristics that allow to differentiate between normal and harmful activity. The representations of *self* referred by Aickelin, Greensmith et al. [61] can be realized in form of binary strings, real-values strings, vectors, multi-dimensional vectors or any other data structure that is appropriate to hold the data determined to be significant in the context of the application purpose.

The decision for a representation of *self* introduces different complexity to the data structure as well as different degrees of accuracy and, thereby, it obviously effects the generation process of the detector set. Besides the original random process for detector generation, Aickelin, Greensmith et al. [61] reference works that try to improve the process by applying other methods such as greedy algorithm to reduce redundancy in detector set, (dynamic) clonal selection from AIS, fuzzy rules, supervised and unsupervised learning from classifier algorithms, and evolutionary algorithms in order to reduce the time complexity of detector generation as well as in order to achieve a better coverage of *self* in the detector set. (Most of these methods are also applied in conventional anomaly detection without employing AIS as introduced in Chapter 2). Aickelin, Greensmith et al. also refer to approaches that use positive selection instead of negative selection to achieve better detection results in terms of coverage as well as in terms of precision in classification (in order to avoid false classifications).

Considering classification, the review provided by Aickelin, Greensmith et al. [61] points out differences in the applied matching methods. The authors compare their efficiency and performance and show strengths and weaknesses of the approaches. Diverse mechanisms are applied in operation phase, from simple exact string matching methods, counting *k* mismatches that classify monitored activity in a threshold-based manner, nearest neighborhood exploration as well as classifier- or cluster-based algorithms, each method having its specific application purpose where it outperforms others. There is no clear statement, which of the approaches reviewed by Aickelin, Greensmith et al. performs best.

As the application purposes considered in the review are solely systems that execute continuously, data generated by these systems can be classified as sequence data or time series, the approaches discussed in [61] are related to defining a window size for the data. Hence, this characteristic also effects the performance of the individual approaches. We will show that for the application domain of the anomaly detection proposed in this thesis we do not have to care about windows sizes due to the characteristics of real-time applications in Chapter 7.

Mainly, the authors point out, that there is a trade-off between scalability of the approaches in terms of size of the detector set, the accuracy of detectors, the time complexity for their generation, and the accuracy and time complexity originating from the applied detection method. Hence, the presented approaches trade between scalability and coverage. Moreover, Aickelin, Greensmith et al. [61] emphasize that current techniques based on the self/nonself discrimination are not able to cope with dynamic and increasing complexity of computer systems as they fail in taking into account that normal and legitimate system usage may change over time. In [61], they point out that *normal* profiles need to be dynamic and adaptable, but none of these approaches is able to overcome this problem because they are relying on a training phase.

In [67], Lay et al. emphasize that real-time systems are becoming more complex and that they face requirements in terms of adaptability and autonomy. Dependability is a challenge as traditional methods are restrictive in this context. Based on this assumption, Lay et al. [67] examine the applicability of AIS to real-time embedded systems. Predominantly, Lay et al. point out the performance of AIS-based systems in terms of anomaly detection by providing easily-computable metrics and algorithms. Nevertheless, AIS-principles belonging to the adaptive immune system such as Self/Nonself Discrimination or Clonal Selection have significant memory and processing requirements which makes an effective use of these techniques impractical being integrated into real-time systems. They propose to apply innate immune principles that are mainly represented by the Danger Theory and the Dendritic Cell Algorithm as they mainly operate on system signals inherently extractable from the system and do not require the processing and memorization of observed behavior patterns.

4.3 Danger Theory-based Anomaly Detection

With [19], Aickelin et al. motivated using the Danger Theory for IDS as it dissolves the limitation of self/nonself discrimination. A number of diverse approaches was developed applying the Danger Theory for anomaly detection. In the review, Aickelin, Greensmith et al. [61] sum up IDS those approaches published until then. The review can be considered to reflect the development and application of the Danger Theory at an early state because at the time this review was published, the community dealing with the Danger Theory just started to grow. Hence, today there exist numerous solutions using the Danger Theory for anomaly detection so that we are also only able to refer to the most important ones that have inspired our work.

One approach referenced by Aickelin, Greensmith et al. [61] and known as the first one published is `cfengine` by Burgess et al. [29, 28]. They apply the Danger Theory for the purpose of a hybrid IDS, combining network-based and host-based, for detecting damage at an initial attack stage. They are making use of danger signals that are raised in case of statistical anomalies in observed event data: on the one hand host-based data such as number of users, number of processes, average utilization, as well as network-based data such as number of incoming and outgoing connections or number of packet loss in the network on the other hand. The danger signals deal as co-stimulation signals that enhance the classification of anomalous system activity and, thereby, strongly impact the reduction of false classification rates. The idea to use danger signals reflecting the system-internal parameters gave us a good inspiration.

Aickelin also contributed to the approach published by Kim, Greensmith, Twycross and

Aickelin [62] dealing with the detection of malicious code execution mainly ensuring compliance with the configured system call policy (in the UNIX program `systrace`). Their idea to use the Danger Theory in their work was motivated because of the problem that system call-based anomaly detection suffers from high false rates. In order to overcome this, they proposed to couple the system calls with its environmental condition and, thereby, build up a context for the system call execution. The Dendritic Cell (DC) is the central component in their approach being responsible for antigen collection and processing of the environmental signals in order to classify the antigen collected. The antigens in the context of this work are subsets of system call sequences executed by the systems. The signals that contribute to the classification of an antigen are defined as: *PAMP* determining a violation in the security policy (signature-based), *danger signal* reflecting high CPU load and/or high memory usage, *safe signal* reflecting continuous normal CPU utilization and memory usage, and the *inflammation signal* expressing the average system load. Multiple DCs examine one set of system calls, each considering an individual partial string, for identifying malicious executions in a population-based manner.

This proposal is the first approach known to us that combines behavioral system and process information with information of the executed system calls. It is not really clear whether the authors have implemented and evaluated this approach. No details about implementation issues are presented so that the paper [62] leaves many open questions: how does the representation of the system call subset looks like? which matching algorithm is applied? how is the classification method implemented? which structure is used for memorizing mature DC? what is the resulting runtime complexity? etc. Nevertheless, the idea proposed by Kim, Greensmith, Twycross and Aickelin addressed many challenges that this thesis is also facing. By combining system call behavior with signals reflecting the system state, a more reliable classification is possible that does not rely on trained data but examines the behavior with respect to its current context. Even if the approach [62] was not intended by its developers to be applied in a self-x environment, the context-related classification builds up the basis for the anomaly detection for self-reconfiguring systems.

Aickelin continued research on Danger Theory-based IDS and the work of his research group was published in the context of an interdisciplinary project called Danger Project (the results of this project - inter alia the Dendritic Cell Algorithm - could be found at www.dangerproject.com until 2012). Together with Twycross, Aickelin developed a software system called `libtissue` that builds up a general framework for implementing and evaluation AIS-based algorithms for anomaly-based intrusion detection [89]. `libtissue` is a library for Linux that provides all building blocks to implement AIS-based systems. It provides wrappers for antigens that represent the structure of entities and for signals reflecting the behavior of entities in the environment of the antigens. It offers functions to be implemented for antigen processing, including collection and presentation, signal processing, cellular bindings, antigen matching as well as immune response. Developers are allowed to program these functions according to their respective problem. Every building block provides a set of parameters to be configured by the user. The architecture of `libtissue` is composed of a *libtissue client* responsible for monitoring the host (processes, operating system, network, etc.) as well as implementing the immune response, and a *libtissue server* containing the algorithms and the storage of antigens and signals. The client collects the data (antigens and signals) and, if required, preprocesses it. Then, it sends the data to the server in order to be processed by the AIS-based algorithm and delivers the classification results back to the client that is able to induce the reaction. The data is stored at the server-side that is considered to be the natural counterpart of tissue. The *libtissue client* and the *libtissue server* are periodically scheduled and communicate via a socket-based connection.

Twycross and Aickelin have evaluated `libtissue` by using the approach of Kim et al. presented above. `libtissue`'s utilization has been demonstrated in [89] on the example of detection of malicious code execution based on system calls and their policies configured in `systrace` in the context with signals reflecting CPU usage. However, the authors emphasize that `libtissue` is applicable to more complex systems as well.

`libtissue` is a very powerful tool as it allows for any kind of AIS-based intrusion detection with diverse points of interest specified as the behavior to be observed in a Linux-like system. As the building blocks are standardized, `libtissue` allows for a systematic analysis of the approaches and makes the different approaches comparable in their performance. Unfortunately, in the context of this thesis, `libtissue` fails to be integrated into a real-time operating system because the behavior of its building blocks has not beed designed for deterministic behavior. One additional problem may arise from the socket-based communication that may cause a further source of unpredictability in terms of timeliness. As it allows many parameters to be configured, it is questionable whether `libtissue` is able to be integrated in a resource restricted platform as those we are addressing here. Because being designed for executing in periodic intervals, an IDS implemented by means of `libtissue` is not able to detect anomalous or even malicious behaviors at the point of time they occur, meaning in a true online manner - an important aspect for us as it was formulated in the requirements introducing this thesis.

Lay et al. in [68] discuss the problem of how to solve "..anomalies in the domain of real-time and embedded systems" and propose to apply Danger Theory. In fact, they build up a schedulability analysis inspired by the Danger Theory that operates on real execution times in order to overcome wasting of resources caused by static analysis techniques only taking into account worst-case execution times (WCET). In their approach, they show that relying on the signals defined by the Danger Theory is more reliable and more efficient than only monitoring behavior for the purpose of detecting deviations.

This short overview is of course incomplete but it points out that there are different viewpoints how to integrate the Danger Theory into a computation system for the purpose of anomaly detection. Based on the objective of the anomaly detection, the Danger Theory allows to specify according behavior (antigen) as well as input signals composed of *danger signal*, *PAMP signal*, *safe signal* and *inflammation signal*. In fact, the Danger Theory seems to be a promising approach applicable for anomaly detection that classifies observed behavior not only based on the structure it exhibits but, moreover, it takes into account the behavior's environmental context by which false classification rates can be reduced.

4.4 Discussion and Summary

System call-based anomaly detection was mainly influenced by Forrest and Hofmeyr who argued that the system call interface is an adequate source of information for enhancing a system's security. By monitoring and analyzing executed system calls, it is possible to detect deviations in program behavior without any specification-based knowledge. Furthermore, system call-based anomaly detection is possible to be integrated into a system without any need to reprogram or recompile application programs. Existing approaches that have been analyzed suffer from complexity in terms of computation time and memory requirements which makes them difficult to be applied in an online working anomaly detection module.

Forrest and Hofmeyr have proposed to use Artificial Immune Systems for overcoming the problem of computational complexity at operation time and expected to deliver a greater level of reliability by reducing development complexity. They proposed to use self/nonself discrimination for anomaly detection and realized this in the context of system call-based analysis. The idea of applying AIS-based approaches for anomaly detection has inspired numerous researchers to follow their idea. Several different approaches for anomaly detection inspired by self/nonself have been developed. However, it turned out that it is challenging to generate an adequate detector set in the training phase in order to ensure a certain degree of coverage. Furthermore, the main drawback related to self/nonself discrimination is its dependency on a training phase in which a *normal* behavior knowledge base is established. The fixed definition of *normal* contradicts the application purpose addressed in this thesis concerning the *normal* behavior as it is expected to change dynamically.

Apart from self/nonself discrimination, Artificial Immune Systems offer the Danger Theory mainly advanced by Aickelin and his co-researchers. Different approaches for diverse application purposes of anomaly detection have been proposed. Especially, the system call-based anomaly detection using principles of the Danger Theory proposed by Kim et al. [62] has inspired the idea of this thesis. Because the Danger Theory enables context-related classification based on the presence and absence of input signals, it is not only related to knowledge about *normal* system call sequences but also allows to classify novel occurring system call sequences within their environmental context. Therefore, it seems to be applicable for self-reconfiguring systems.

One main question that is left open by all these Danger Theory-based approaches is the implementation the pattern matching process and the storing of the system call sequences as representatives of the antigens. For contiguous processes this question is challenging and often mechanisms such as sliding windows are utilized to isolate an antigen from the entire data sequence. This is different for real-time systems as real-time applications exhibit valuable characteristics that enable to enclose the behavior sequences by their periodic execution. Hence, different kinds of pattern matching mechanisms and storing structures have to be considered for this purpose. Foundations into this topic are provided in the next chapter.

CHAPTER 5

Online Pattern Matching

5.1 Introduction

In Chapter 1.1, requirements for the Anomaly Detection approach have been formulated. One of the major requirements is the evaluation of behavior patterns in terms of their previous occurrences. This aspect defines two subtasks to be solved:

- every occurring behavior sequence has to be stored
- every new behavior sequence has to be matched against the set of previously executed behavior sequences.

In fact, these tasks address the problem of finding an adequate data structure for storing and the problem of pattern matching.

Pattern matching mainly deals with the question [53]: "Given a string P called the *pattern* and a longer string T called the *text*, the exact matching problem is to find all occurrences, if any, of pattern P in T."

Definition: Substring

A sequence $P = p_1...p_n$ is a called subsequence or *substring* of the string S, if there exists a string $X = s_l..s_{l+n-1}$ in $S = UXW$ (U,W being strings, also possibly empty) with $p_k = s_{l+k-1}$ for every $k : 1 \leq k \leq n$.

Generally, pattern matching consists of two phases: the preprocessing of the text T and the matching process of pattern P. Referring to the formulated tasks, the preprocessing of the text is related to the storing of an executed behavior sequence as an executed pattern must be made available for matching with behavior sequences that will follow. The matching process is performed on the newly arriving behavior sequences that builds up the pattern P as the matching object.

In order to make the pattern matching applicable in a real-time system, the task of pattern matching is restricted in terms of being deterministic in run-time and memory requirements. Hence, the execution time as well as the memory load produced by the pattern matching must be bounded. Furthermore, to ensure real-time performance, anomalies must be detected as soon as they occur which leads to a pattern matching that must be performed online.

There are approaches to perform online (sub-)sequence pattern matching in order to detect - exact matching - common substrings between sequences. Some of them use sliding window techniques to compare and find matching substrings. In [77], for example, Mueen and Keogh evaluate the first approach for online discovery of exact patterns in time series data that requires only linear runtime with respect to the windows size. However, the main drawback of this approach is that it only operates on recent history data as the search space for pattern discovery, and thereby discards older data as well as the hitherto discovered occurring patterns. Above all, the efficiency of sliding windows techniques, usually, suffers on the decision of windows size. This can be either too small to detect complete recurring patterns, or it is chosen too large which increases the runtime complexity of the matching process.

Apart from sliding window techniques, approaches such as *Dynamic Time Warping* are applied for strings of (different) finite lengths with the objective to detect all common substrings including their number of occurrence. In *Dynamic Time Warping*, string matching problems are solved by means of grids [26] that match a query S against a sequence of data points T by searching for $s_i = t_j$ and recording the distances of any $s_i - t_j$. The required time complexity is $O(n \cdot m)$ with n the length of S and m the length of T. The task of the pattern matching process in the context of our anomaly detection approach is to detect whether the currently occuring query S has already occurred in history. Then, T shall contain the entire history of behavior sequences. Even if the runtime complexity is bounded by n and m, T is extended by each execution of a task instance and, consequently,the length of T is increasing with each execution of a task instance which makes m to become unbounded. Besides the performance of those approaches, the challenge remains how to store all discovered common patterns in a compact and bounded manner. These two arguments make *Dynamic Time Warping* not applicable to the present online anomaly detection problem.

The objective of this chapter is to discuss online pattern matching approaches that are applicable for anomaly detection in real-time systems. In the context of this thesis, Suffix Trees have been identified to be best suitable for matching as well as storing. Therefore, in the remaining part of this chapter, Suffix Trees are introduced in detail showing why the characteristics of Suffix Trees are best suitable for the problem solved by this thesis. The content of this section mainly refers to Gusfield [53] (Part II, Chapter 5 and 6) and to Martin Kay's lecture notes [59].

5.2 Suffix Trees

Suffix Trees belong to the fundamental data structures in computer science even though they did "not make it into mainstream of computer science education..." [53]. For the introduction of Suffix Trees, some definitions are helpful beforehand:

Definition: Suffix
For a string S of length m, every $S[i..m]$ with $1 \leq i \leq m+1$ is a suffix of S. $S[(m+1)..m]$ is defined as the empty suffix.

Definition: Prefix
For a string S of length m, every $S[1..i]$ with $0 \leq i \leq m$ is a prefix of S. $S[1..0]$ is defined as the empty prefix.

Refering to these definitions, every substring P of S is a prefix of the suffix of S ($S = UPW$, with PW building a suffix of which P is a prefix).

A Suffix Tree is a tree data structure that represents a string S of final length m in such a manner that, for every suffix of the string S, it contains one unique path starting from root.

Definition: Suffix Tree [53]
A suffix tree τ for an m-character string S is a rooted directed tree with exactly m leaves numbered 1 to m. Each internal node, other than the root, has at least two children and each edge is labeled with a nonempty substring of S. No two edges out of a node can have edge-labels beginning with the same character. The key feature of the suffix tree is that for any leaf i, the concatenation of the edge-labels on the path from the root to lead i exactly spells out the suffix of S that starts at position i. That is, it spells out $S[i..m]$.

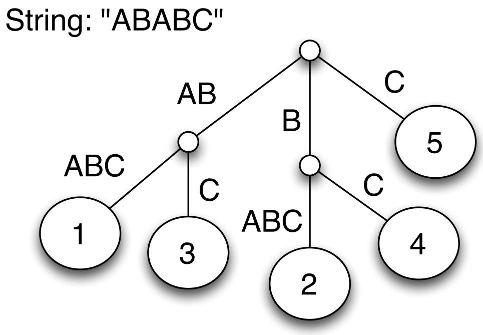


Figure 5.1: Suffix Tree for String S= ABABC

Figure 5.1 shows an example for a Suffix Tree for the string $S = ABABC$. Suffixes of the String S are $ABABC$, $BABC$, ABC, BC and C. Every suffix is represented by a unique path in the tree which is identified by the labels on the edges. The labels on the edges of a path constitute - by concatenation - the suffix of S. The Suffix Tree does not represent the empty suffix of a string.

Suffix Trees are build to solve a number of string problems, such as the exact string matching problem, the substring problem, the longest common substring of two strings as well as the DNA contamination problem. Suffix Trees are well-known to solve these problems in linear time (see [53]). Moreover, according to Gusfield [53], they build a bridge between *exact matching problems* and *inexact matching problems*. The applicability of Suffix Trees to those problems will become clear when considering the characteristics that Suffix Trees provide.

5.2.1 Basic Characteristics

The basic assumption in context of Suffix Trees is that the string S, for which the Suffix Tree is constructed, is composed of symbols of a finite alphabet. For the examples in this Chapter, we use the alphabet= $\{A, ..., Z\}$ for the string characters.

Deriving from the definition, the Suffix Tree for a string S of length m has the following properties [59]:

1. the Suffix Tree has exactly m leaves numbered from $1..m$
2. every node has at least two children (except the root node)
3. each edge is labeled with a non-empty substring of S

4. no two edges starting out of a node will have string-labels beginning with the same character

5. the string obtained by concatenating all the string-labels on one path from the root to leaf i spells out the suffix $S[i...m]$, $i = 1..m$

However, the last property cannot be explicitly guaranteed for strings with a suffix completely matching a prefix of another suffix. An example for such a string is $ABCBC$. The suffix $S[4..5] = BC$ is a prefix of the suffix $S[2..5] = BCBC$ and the suffix $S[5..5] = C$ is a prefix of the suffix $S[3..5] = CBC$. Figure 5.2 shows the Suffix Tree for string $ABCBC$, having no explicit paths for the suffixes $S[4..5] = BC$ and $S[5..5] = C$ and hence no leaves at the positions 4 and 5. Such a Suffix Tree is called *implicit* Suffix Tree as it does not provide an explicit unique path for every suffix of the string ending in a leaf node. Nevertheless, it encodes all suffixes of the string.

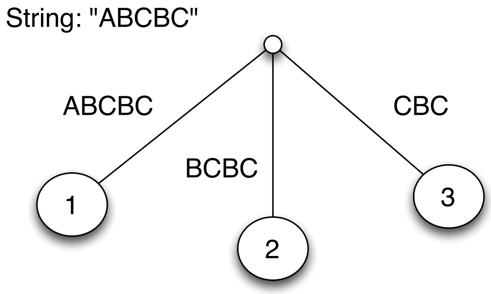


Figure 5.2: Implicit Suffix Tree for String S= ABCBC

In order to ensure one unique path for every suffix ending in a leaf node, a common practice is to extend every string by a unique *termination* symbol that is not in the original alphabet. Usually, the symbol $ is used as the termination symbol and is added at the end of a string. Extending the example string $ABCBC$ by the termination symbol, we obtain $ABCBC\$$ and the Suffix Tree as illustrated in Figure 5.3. Thereby, we obtain a Suffix Tree containing a unique path for every suffix with $m = 5$ (the length of the string) leaves. For the further explanations in this chapter, every string is assumed to be ending with the termination symbol $, even if not explicitly denoted, in order to ensure the existence of *true* Suffix Trees for the strings.

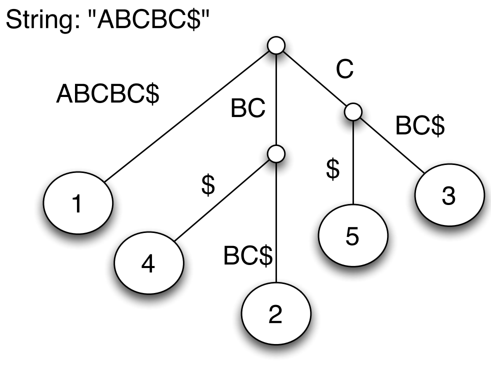


Figure 5.3: Suffix Tree for String S= ABCBC$ extended by the termination symbol $

For solving the pattern matching problem, in particular, the substring problem, Suffix Trees provide the following characteristics (assuming the Suffix Tree was build up for the text T and the pattern P to be found in T):

- If pattern P is a substring of T, then P is a prefix of a suffix of T and, consequently, a path in the Suffix Tree exists, that starts from root node and is labeled by P (by concatenating the labels of the edges). The path determining the pattern P does not necessarily need to end at a leaf node. (E.g. the pattern $P = BCB$ in the text $T = ABCBC\$$. Hence, by matching the pattern with the labels in the Suffix Tree in a symbol-by-symbol manner starting from root, it can be verified whether the pattern P is a substring of T. Furthermore, by the symbol-by-symbol comparison, the longest common substring can be identified, even if the pattern P is not (completely) a substring of T.

- If a pattern P is a substring of T, then the number of occurrence of pattern P in T can be directly derived from the structure of the Suffix Tree. Pattern P is found in the tree as described above and its path in the Suffix Tree is identified by the matching path' labels. The subtree below the last match contains leafs that denote the starting location of P in T. Hence, the number of leaves of that subtree is equal to the number of occurrence of pattern P in T. In the Suffix Tree for $T = ABCBC\$$ (see Figure 5.3), the subtree below pattern $P = BC$ has two leaves identified by the positions 2 and 4. Hence, $P = BC$ occurs twice in $T = ABCBC\$$.

Because of the structure of the Suffix Tree, the pattern matching process is simple: following the path in the Suffix Tree that is labeled by the pattern. (If no path exists, the pattern is not contained.) According to Gusfield [53], by assuming a finite alphabet, the workload to detect the accordingly labeled edge descending from a node can be considered as consuming constant time. Hence, the time costs required to match the pattern against the path labels is proportional or linear to the length n of P, but independent of m, the length of the text T. Generally, the search/matching process in Suffix Trees requires $O(n)$ time.

The problem of counting the number of occurrences, initially, involves the search process as counting can only be performed on patterns that exist in the Suffix Tree. Additionally, it requires to traverse the subtree below the path of the matching pattern in order to count the number of leaves. In worst case, the subtree may have the maximum number of leaves of m (length of T). Therefore, the time required for counting the number of occurrences of a pattern P in a text T results in $O(n + m)$.

However, before being able to execute the matching process, the Suffix Tree has to be built up which happens in the preprocessing phase. First, a trivial approach for the construction method is provided. Then, the approach introduced by Ukkonen is illustrated as it is famous for constructing the Suffix Tree in linear time and best suitable for the approach discussed in this thesis.

5.2.2 Basic Construction Method

In order to ensure a clear terminology and understanding, we first provide some definitions:

- the root node is denoted as r
- every inner node is identified by a letter $\{a, ..., z\} \backslash \{r\}$; the set of nodes is denoted by N
- leaf nodes are labeled by the starting position of the suffix in the text T

- an edge is denoted by $e_i = (u, v)$ if e_i connecting u and v, with $u, v \in N$ and i the index of the edge
- the Suffix Tree for string S is called $\mathcal{T} = \mathcal{T}(S)$

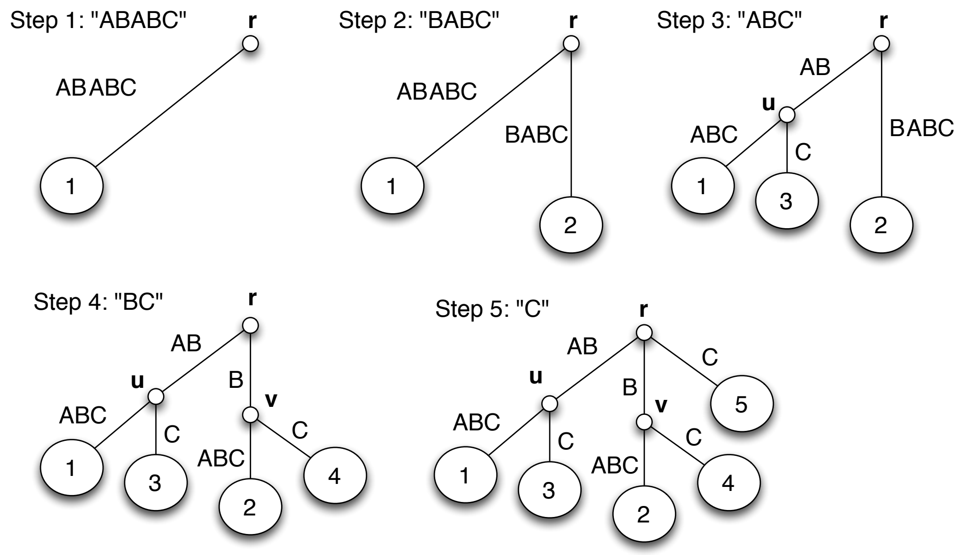


Figure 5.4: Steps performed in the Naive Suffix Tree Construction Method for String S = ABABC

The basic construction method for Suffix Trees described in this section is often called the *naive* algorithm. In the *naive* algorithm, the Suffix Tree is constructed in an incremental way, by adding the m suffixes $S[i..m]$ one after the other, for i increasing from $i = 1$ to m. The Suffix Tree obtained after step i is denoted as $\mathcal{T}_i$.

To support the generic description following, the step-wise construction is illustrated in Figure 5.4 for the example string S= ABABC (as this string does not lead into an implicit Suffix Tree, we left out the termination symbol for the purpose of simplicity).

Initially, the Suffix Tree $\mathcal{T}$ consists of only the root node r. As a fist step, a single edge is entered for $S[1..m]$. The edge is labeled by the string $S[1..m]$ and ends at the leaf labeled 1 (see step 1 in Figure 5.4). $\mathcal{T}_1$ is constructed. The Suffix Tree $\mathcal{T}_{i+1}$ is (incrementally) constructed on the basis of $\mathcal{T}_i$. In any $i + 1$th step, the suffix $S[i + 1..m]$ is extending the Suffix Tree $\mathcal{T}_i$ in order to obtain $\mathcal{T}_{i+1}$ by the following procedure: To insert the suffix $S[i + 1..m]$ into $\mathcal{T}_i$, first it is necessary to find the longest path from root whose labels match the prefix of $S[i + 1..m]$. The last matching character is $j : i + 1 \leq j \leq m$ or $j = i - 1$ if there is no match. The Suffix Tree $\mathcal{T}_i$ must be extended by $S[j + 1..m]$ behind the character that points to the last match.

If there is no path matching the prefix of $S[i + 1..m]$, the pointer to the last matching character points to the root node and requires the complete suffix to be added below that. This is done by creating a new leaf j and an edge connecting it that is labeled by that suffix. Referring to the example in Figure 5.4, at step 2 there is no match with an existing path for $S[2..5] = BABC$. Hence, a new leaf labeled 2 and an edge $(r, "2")$ is created.

If there is a matching path for a prefix of $S[i + 1..m]$, the last matching character $S(j)$ is either ending at the end of an edge label - ending in a node - or is ending in the middle of a label. In the first case, the last matching character is the ending of an edge label of a node v so that the Suffix Tree $\mathcal{T}_i$ has to be extended by $S[j+1..m]$ below the node v by creating a new leaf $j+1$ and an edge connecting $(v, j+1)$ having the label $S[j+1..m]$. In the latter case, the last matching character is in the middle of an edge label of edge (u, w). The edge is split behind the character j by introducing a new node v' resulting in two edges (u, v') and (v', w). Accordingly, the original label of the edge (u, w) is split also resulting in the label $S[x..j]$ (x being the starting position of the edge label $i+1 \leq x \leq j$ for edge (u, v') and the remaining part of the original edge label (after the jth character) now labeling edge (v', w). As v' identifies the path of the prefix $S[i+1..j]$, $S[j+1..m]$ is added below the node v' by creating a new leaf $j+1$ and an edge connecting $(v', j+1)$ having the label $S[j+1..m]$.

In our example, at step 3, the suffix $S[3..5] = ABC$ matches the label of edge $(r, "1")$ in the first two characters. The edge is split and node u is introduced and extended by the leaf 3 and the edge $(u, "3")$. At step 4, the suffix $S[3..5] = BC$ matches in the first character of edge $(r, "2")$ which leads to a split and the introduction of node v the according extension.

The naive algorithm requires $O(m^2)$ time to construct the Suffix Tree for a string of length m. The required number of nodes (in worst case) is $2m$ (1 root node + m leaf nodes + $(m-1)$ inner nodes in worst case), while the space requirement for the labels is again $O(m^2)$.

5.2.3 Construction by Ukkonen's Algorithm

Even though, the first linear-time algorithm for constructing a Suffix Tree was presented by Weiner [94] in 1973, Ukkonen's Algorithm, which first was presented in 1995 [90], is more famous and has become generally known as it is better understandable and easier to implement (see [53]). Besides its linear-time complexity, one of the main important property of Ukkonen's algorithms is that it works online.

Mainly, Ukkonen's algorithm builds up the Suffix Tree for the prefixes $P_i = S[1..i]$ of a string $S[1..m]$ incrementally, starting at the prefix $P_1 = S[1..1]$ to construct $\mathcal{T}_1$ and successively extending the Suffix Tree by entering the prefix P_i (meaning $P_1 = S[1..1]$, $P_2 = S[1..2]$,... $P_k = S[1..k]$, .. $P_m = S[1..m]$) to achieve $\mathcal{T}_i$, for i from 1 to m. Every prefix P_i is processed within a so called *phase* in Ukkonen's algorithm so that the entire process is divided into *m phases*.

In each phase i, the prefix P_i is again processed incrementally by means of *j extensions*. The extension j, j from 1 increasing to i, is concerned with enclosing the suffix of $R_j = S[j..i]$ of the prefix P_i into the current Suffix Tree in order to obtain $\mathcal{T}_i$. That means, that for *extension j* in phase i, the algorithm extends the path for $S[j..i-1]$ (in $\mathcal{T}_{i-1}$) by symbol $S(i)$, of course, unless it is not already there. The path of $S[j..i-1]$ is presupposed to exist in the Suffix Tree as it was enclosed in the previous phase $i-1$.

For the extension of symbol $S(i)$ (in *extension j* of phase i) three rules are specified:

Rule 1 If $S[j..i-1]$ is the label of an edge that ends at a leaf node, the edge label is updated and $S(i)$ is added to the end of that edge label.

Rule 2 If the path $S[j..i-1]$ continues below $S[j..i-1]$ by at least one path and none of the continuing paths' labels starts with $S(i)$, a new leaf edge is created starting from the end of $S[j..i-1]$, with the leaf label j and the edge label $S(i)$. If $S[j..i-1]$ ends in the middle of an edge, then, before adding the leaf edge, a new inner node is inserted as a source of the leaf edge.

Rule 3 If some path from $S[j..i-1]$ starts with the character $S(i)$, then there is nothing to extend.

Figure 5.5 illustrates the results after executing each of the m phases for the example string S= ABABC. In phase 1, the prefix $P_1 = A$ is inserted into the empty Suffix Tree resulting into $\mathcal{T}_1$. As $i = 1$, there is only one extension for $j = 1$. In phase 2 for the $P_2 = AB$, the extension 1 handles the suffix $R_1 = S[1..2] = AB$ by applying **Rule 1**. The extension $j = 2$ handles the suffix $R_2 = S[2..2] = B$ by adding the leaf edge labeled 2 as defined by **Rule 2**. In the phases 3 and 4, all *extensions* but the last ones (when $j = i$) require only the adding of the edge labels as applying **Rule1**, while for the last *extensions* $j = i$, the characters $S(i)$ are already contained in the $\mathcal{T}_{i-1}$ and therefore, do not require any action as defined in **Rule3**. In phase 5, the algorithm makes use of **Rule3** and inserts new inner nodes and the according leaf edges for the suffixes of *extensions* $j \geq 3$ (meaning when including the suffixes $R_3 = S[3..5] = ABC$, $R_4 = S[4..5] = BC$ and $R_5 = S[5..5] = C$). In fact, the algorithm appends $S(i)$ to every suffix $R_j = S[j..i-1]$ of each prefix $P_{i-1} = S[1..(i-1)]$ of S.

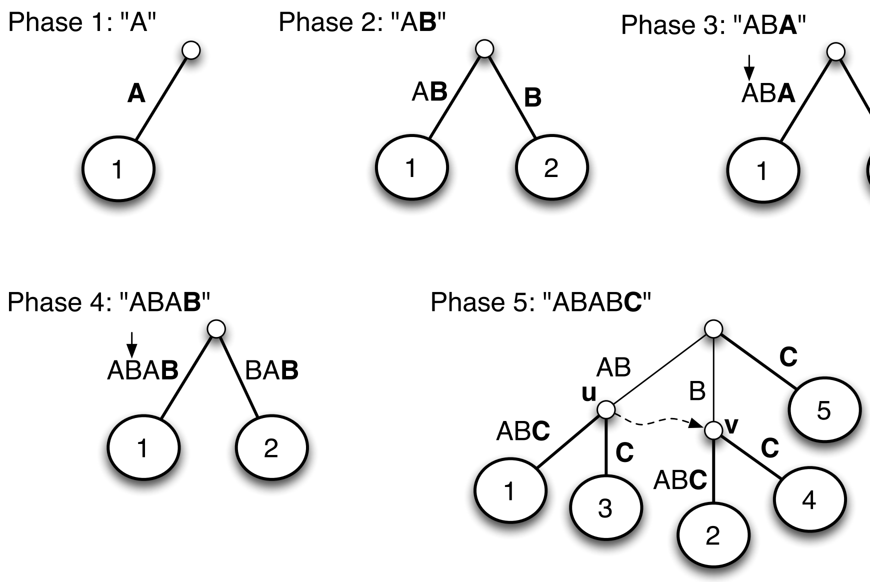


Figure 5.5: Steps performed in the Ukkonen's Suffix Tree Construction Method for String S = ABABC

By processing prefixes of string S, the endings of the suffixes of the string are not marked explicitly. Actually, Ukkonen's algorithm builds up an implicit Suffix Tree $\mathcal{T}_m$ for a string S. After completing all phases of the algorithm, the resulting implicit Suffix Tree $\mathcal{T}_m$ is converted into a true Suffix Tree: this is performed by adding the *termination* symbol \$ to the end of the string S and executing Ukkonen's algorithm for that prefix S\$ once again. The conversion of an implicit Suffix Tree $\mathcal{T}_m$ into a true Suffix Tree is done in $O(m)$ time.

To build up this tree by Ukkonen's algorithm described so far, $O(m^2)$ time is required. Ukkonen introduces Suffix Tree representation elements that in combination are exploited by implementation tricks to speed up the algorithm into a linear time bound. The Suffix Tree representation elements are:

1. **Suffix Links**

   Definition: Suffix Link [53]:
   Let $x\alpha$ denote an arbitrary string, where x denotes a single character and α denotes

a (possibly empty) substring. For an internal node v with the path-label $x\alpha$, if there is another node $s(v)$ with the path-label α, then the pointer from v to $s(v)$ is called *Suffix Link*.

Such a *Suffix Link* is shown in Figure 5.5 between node **u** and node **v**.

Suffix Links reduce the effort required to traverse the tree for finding the path for $S[j..i]$ from the path that determines $S[j-1..i]$ which is required for any incrementation of the *extension* in a *phase*. Furthermore, searching the path for $S[j..i]$ from path for $S[j-1..i]$ by the *skip/count trick* introduced by Ukkonen (see [53] for further details) becomes independent from the number of characters of $S[j..i]$, but, instead, it gets a function proportional to the number of nodes.

2. **Edge-Label compression**

   The required amount of space for a Suffix Tree is determined by the number of nodes, the number of connecting edges, and, in particular, the space required for the edge labels. As the Suffix Tree contains one path for each suffix (by concatenating its labels), the worst case space requirement for the labels is $O(m^2)$ when using the original label representation.

   Edge-label compression, however, provides an alternative edge-label representation. Instead of labeling an edge by the characters of the substring $S[p,q]$, edge-label compression replaces the labels by a pair of indices (p,q). Since the string is stored separately in S (e.g. in an array), the edge label points to the substring's starting point in the string S by p and to the substring's ending character by the value of q. The extension rules for the Suffix Tree construction are accordingly adjusted to this labeling scheme.

   The *edge-label compression* reduces the space requirement of an edge to a constant value. Hence, the total space requirement for edge labels is reduced to $O(m)$, being proportional to the number of edges.

Applying *edge-label compression* combined with using *Suffix Links* for path-finding and some further implementation tricks to reduce computation effort in the *extensions* (described in detail in [53]) yields to a construction time for the Suffix Tree of $O(m)$. For details of the proof for the achieved speed-up, see [53].

Because, in each phase i, the algorithm proceeds one further symbol (the ith symbol) of the string, the construction method is called to proceed the string in a *symbol-by-symbol* manner. From this, its online property is derived: At any point of time, Ukkonen's algorithm can build up the Suffix Tree for the prefix of the string that occurred up to now, without having knowledge about the remaining (yet unknown) suffix of the string. Hence, the Suffix Tree can be extended by each occurrence of the next symbol of the string, which actually defines *online processing*.

5.2.4 Generalized Suffix Tree

A Suffix Tree, as defined up to now, is a data structure to represent one single string S. However, Suffix Trees can be used to represent multiple strings S_1, S_2,...S_k in one data structure. These Suffix Trees are called *Generalized Suffix Trees*.

The construction method for *Generalized Suffix Trees* is based upon the construction method for an ordinary Suffix Tree. First, the Suffix Tree $\mathcal{T}_{S_1}$ for string S_1 is built up. For the second string S_2, the Suffix Tree $\mathcal{T}_{S_1}$ is extended by S_2. The ordinary construction method (e.g. Ukkonen's algorithm) is used for this purpose in a modified manner. For every phase i, it is first checked whether P_i is already contained in the tree. If P_i exists in the tree, then phase i for S_2 can be skipped. The resulting Suffix Tree contains all suffixes of the strings S_1, S_2,...S_k it includes.

If the purpose of application requires a distinction between the individual strings, the representation of the suffix tree can be adjusted by the following means:

- Each string S_i may be equipped with an individual termination symbol $\$_i$. Then every suffix of S_i is represented by an own path leading into its individual leaf. **Or:**
- If one common termination symbol is used for all strings, an adjustment of the leaf labels is required: leaf labels hold the starting positions of the suffix in a string. In case of multiple strings, the label is extended into a tuple composed of starting position of suffix and string ID. Furthermore, for each occurrence of the suffix within a string S_i, one label entry associated with that suffix is added to the leaf label. **And:**
- Applying label compression in *Generalized Suffix Trees* requires an extension of the string ID in the edge label entries.

For *Generalized Suffix Trees*, the construction time is proportional to the sum of the string lengths and results in $O(|S_1| + |S_2| + ... + |S_k|)$. If the strings S_1, S_2,...S_k contain equal substrings, the construction time, in fact, will be reduced by any common substring as these extension phases can be omitted in the construction process

5.2.5 Suffix Tries

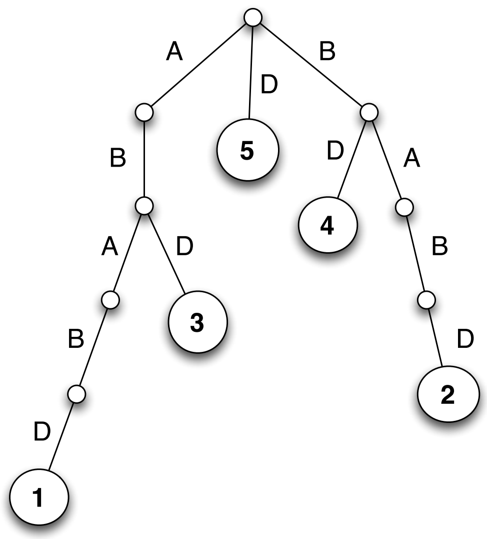


Figure 5.6: Suffix Trie for the string $ABABD$.

A data structure closely related to Suffix Trees are Suffix Tries.In contrast to Suffix Trees, in Suffix Tries, each edge is labeled by only one character of the suffix. Consequently, Suffix Tries require one edge/node per character of a suffix path. Therefore, in Suffix Tries, nodes that do not need necessaritly to have at least two children but can exist with only one child edge. Figure 5.6 shows a Suffix Trie for the string $S = ABABD$.

A Suffix Trie is less compact than a Suffix Tree, as it needs $O(m^2)$ nodes, equal to the sum of characters of all suffixes of S. Suffix Tries can be easily transferred into Suffix Trees in linear time by joining all edges of nodes that are not branching and replacing them by one single edge whose label is composed of the concatenated characters of the joined edges. Joining the edges of non-branching nodes results in a structure containing only branching inner nodes (besides the root node and the leafs) and leads to the Suffix Tree.

Apart from the space requirement, Suffix Tries have the same characteristics as Suffix Trees, especially in terms of the ability of finding matching substrings as well as counting the occurrences of substrings.

5.3 Summary

Numerous pattern matching methods exist in literature. However, their applicability with respect to the requirements formulated in this thesis are limited. Especially, their ability of being integrated into an online process is challenging with the restrictions in terms of determinism in runtime and resources defined by real-time systems. Suffix Trees offer properties that are best suitable for the application purpose addressed here. They provide bounded preprocessing and matching methods that work in an online manner with bounded runtime allowing to process a sequence in a symbol-by-symbol manner. At the same time, in forms of Generalized Suffix Trees, they provide a data structure that can hold multiple sequences and thereby enable to store history data which size - by nature of Suffix Trees - is bounded with respect to the sequences' lengths.

We exploit the characteristics of Suffix Trees for building up the Knowledge Base of our anomaly detection. We adjust the concept of Suffix Trees by some variations and extensions of their basic specifications. These variations and extensions will be discussed in the conception part of this approach in Chapter 7 as well as in the part introducing the implementation of the proposed approach in Chapter 8.5.

CHAPTER 6

ORCOS - Organic Reconfigurable Operating System

Even though the approach for anomaly detection worked out in this thesis is designed to be applicable on any real-time operating system (with specific characteristics that will become more clear in Chapter 7), we have implemented and integrated it into an exemplary real-time operating system to establish a basis for the evaluation of the approach. The real-time operating system ORCOS (Organic ReConfigurable Operating System) [2] has been developed at the University of Paderborn and I was one of the main persons to push ahead its development. This operating system is highly customizable and was designed for reconfigurable embedded systems that implement self-x capabilities and was used in a couple of other research projects. This chapter basically relies on the ORCOS documentation in [2] and provides an overview of the concept and design of ORCOS that build the foundations for the implementation of the anomaly detection.

6.1 ORCOS Design and Architecture

ORCOS is designed as a hybrid kernel architecture which is composed of operating system modules:

- HAL
- Systemcalls
- Processes
- Scheduler
- Memory Management
- Filesystem
- Communication
- Power Management

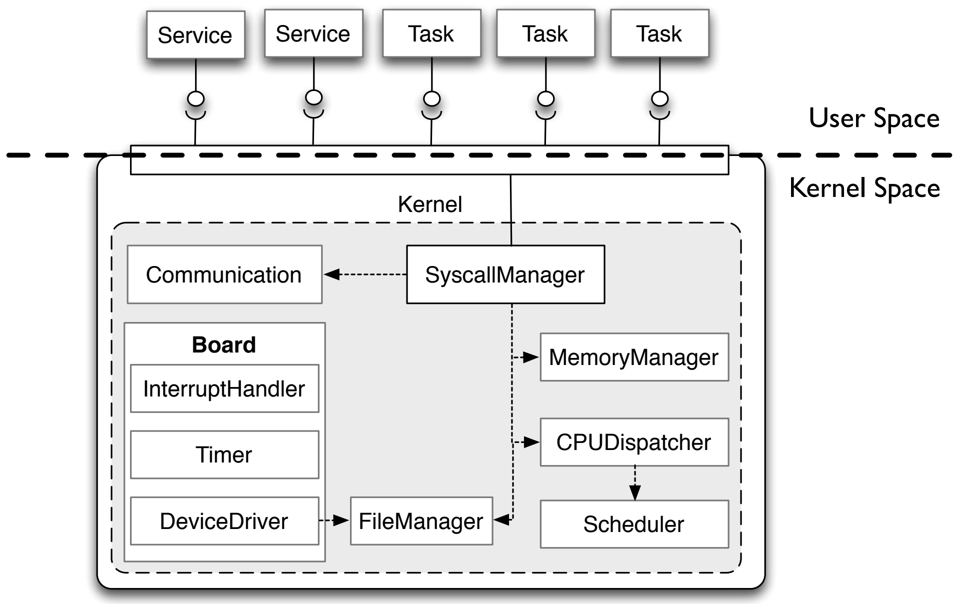


Figure 6.1: ORCOS Architecture: separation of user space task and services and kernel space kernel modules

The ORCOS modules are strongly separated from each other and build up an architecture that is illustrated in Fig. 6.1.

Like any operating system, ORCOS is an interface between applications and the underlying hardware. Application tasks and the kernel are clearly separated in ORCOS by assigning application tasks to the user space and the operating system kernel modules to the kernel space. This ensures that the kernel code and memory space are protected from the user code. For the applications, ORCOS provides the System Call Interface as the only interface between applications tasks and the kernel functions. To ensure functionality of the operating system, the basic modules *System Call Interface*, *Memory Manager* and *Filemanager* are absolutely required and therefore, they are permanently integrated in the operating system. All other modules are optionally configurable. The specific kernel modules are described in more details in section 6.3.

The ORCOS lowest layer module is the HAL (Hardware Aabstraction Layer) and it forms the basis for portability of ORCOS to multiple hardware platforms. The HAL hides implementation differences between different hardware architectures by providing a dedicated interface to hardware functionalities for the kernel.

Up to now, ORCOS supports a number of architectures such as

- PowerPC405
- Sparc Leon 3
- ARMv4(t) and above

Furthermore, ORCOS runs in an emulated manner using QEMU[4] for the PowerPC405 architecture.

ORCOS is designed to be a complete object-oriented operating system. Besides hardware-dependent functions, that are implemented in Assembler and C, the entire operating system is programmed in C++. All required functions are implemented within ORCOS itself so that no external (C++) libraries are employed. A clearly structured self-developed operating system library reduced to the essential functions was included into ORCOS. By this, we could minimize the size of the operating system that in a small OS binary memory footprint.

6.2 Offline Configurability

With the ORCOS configuration system the developer is able to configure the whole system within a single configuration file. All ORCOS modules and tasks are individually configurable at compile time. The configuration system allows to set options that control parts of the system behaviors as well as settings of the tasks that should be loaded on the target device. Therefore, ORCOS offers an XML-based language named Skeleton Customization Language (SCL) (see Documentation on [2]) to simplify the process of the ORCOS configuration. By Skeletons, developers can decide which implementation of a class/module and which member type to integrate into the operating system kernel. Valid dependencies between the single module implementations are specified and stored in a central list. At compilation time, the defined SCL configuration file is verified and validated against the predefined dependencies and parameter settings.

With SCL, it is possible to configure an operating system that is fully customized to the requirements of the application and the target platform. ORCOS' small footprint and and its highly flexible customization ensure the applicability of ORCOS on strongly resource-restricted embedded systems.

6.3 Operating System Modules

ORCOS is composed of modules that are individually configurable at compile time (see section above). This sections introduces the main concepts and implementation principles of the particular ORCOS modules.

6.3.1 Task Management and Scheduling

ORCOS implements processes as tasks that can have multiple execution entities. These execution entities are threads which are represented in ORCOS by multiple thread classes available for different objectives and configurations.

Threads

A task in ORCOS is composed of threads. Each task at least owns one threads as, in ORCOS, threads are the scheduling entities. ORCOS offers several classes with different characteristics for thread implementation, such as the basic class `Thread` or a specific class for real-time threads named `RealTimeThread`, etc. A thread contains all the needed information required for the management of the thread as well as its execution. The execution to threads is managed by the Scheduler. At any point of time of a thread's lifetime, it is always assigned to a state. The ORCOS thread state model (illustrated by Fig. 6.2) is related to the common 5-state model for real-time processes (see Stallings [85]).

As threads in ORCOS are the executing entities, each thread is assigned its own stack and context. The size of the stack is individually configurable.

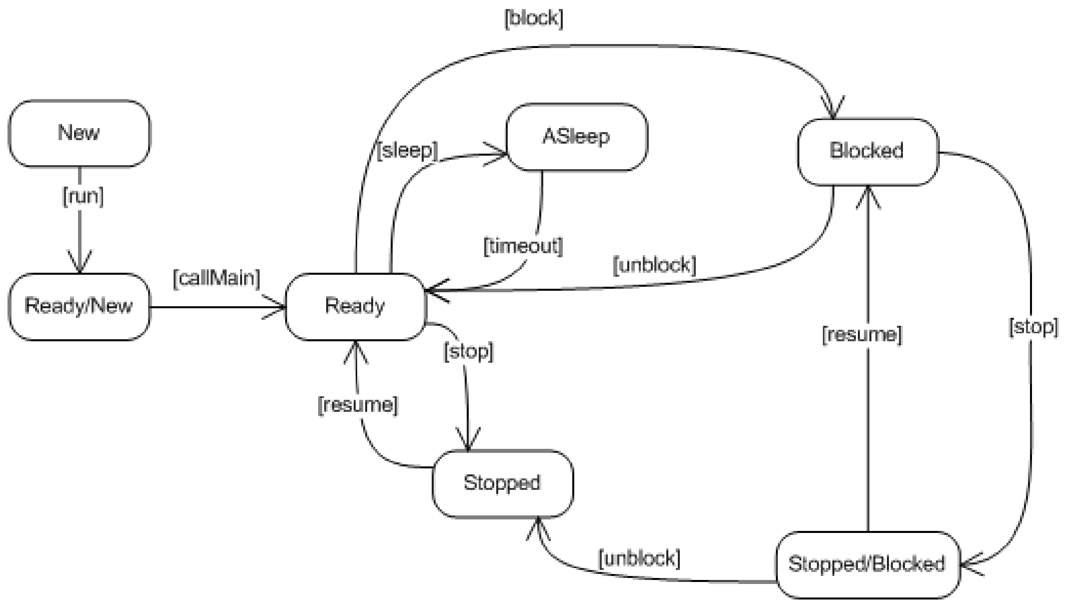


Figure 6.2: Thread state model implemented in ORCOS; Source: [2]

Tasks in ORCOS (and their according threads) can be real-time task or non real-time tasks. When having real-time constraints, a thread will be derived from the `RealTimeThread` class which is extended by attributes like execution time and deadline.

Workerthreads

For operating system workload, ORCOS introduces a specific type of task at the kernel side: the Workertask. There exists exactly one system-wide workertask that usually holds multiple workerthreads. Workerthreads take over kernel activities that appear sporadically as well as in a time-triggered or periodic manner. They are executed in such a way that they are scheduled like any other thread by the system scheduler. The concept of workerthreads makes it possible to preserve predictability of the kernel activities and hence to ensure the predictability of the entire system.

Workerthreads can be applied in three different field of applications:

1. **Asynchronous Call to Functions:**

   Asynchonous calls to functions mainly refer to (but are not restricted to) emerging hardware interrupts. ORCOS therefor follows the standard technique of a real-time operating system (see Buttazzo [30]). Whenever a hardware interrupt arises (if not deactivated), the kernel interrupts any execution. Even if interrupts are arising unpredictably, the OS must remain predictable. Therefore, the processing of the interrupt is implemented within workersthreads containing the interrupt handling routine Then, the interruption by a hardware interrupt is minimized as it takes only the time required to activate the workerthread. (In this context, the number of these interruptions by hardware interrupts can be estimated based on worst case considerations). The workerthread, being inserted into the scheduling queue, is then scheduled according to its priority and the applied policy, so that high-priority tasks will not be preempted due to hardware interrupt handling.

2. **Timed Calls to Functions:**

   Some kernel methods need to be called in a time-triggered manner after a given time interval. Workerthreads implement the calling of these methods.

3. **Periodic Calls to Functions:**

   Periodic Calls to Functions extend the idea of Timed Calls to Functions as the timed function call is assigned an according period. Workerthread for periodic function calls behave like any other periodic thread.

Like ordinary threads, workerthreads possess their own stack with configurable stack size. However, as being kernel-side threads, the memory reserved for the workerthread stacks accumulates to the entire memory usage of the kernel and hence, effects the memory footprint of the OS. Therefore, system designers have to carefully decide how many workerthreads are really required to be configured into the OS kernel in order to satisfy the functional requirements as well as the intended small OS footprint.

Task Structure

A task to be executed in ORCOS has to comply to the ORCOS specification and requires configuration by SCL. Task attributes that are required for task management are stored in the task's `TaskTable`:

- `task_start_addr`: defines the physical start address of the task's memory space
- `task_entry_addr`: defines the logical entry function address
- `task_heap_start`: defines the physical start address of the heap
- `task_heap_end`: defines the first physical address that does not belong to the heap ;it is the last possible entry for a thread's stack
- `task_data_end`: defines the first logical address that is no data of the task anymore (`.text`, `.data`)
- `task_thread_exit_addr`: defines the logical address of the `thread_exit` method inside the task

Sec. 6.3.2 provides a description on the memory layout of a task integrated into the entire memory layout.

Furthermore, additional task attributes are required in terms of (real-time) scheduling . These scheduling parameters are extending the task table structure after being configured by SCL and compiled. They include attributes such as:

- period
- deadline
- execution time
- phase

After configuration and compilation process of the task, these configured attributes will be handed over to the according ORCOS modules (Task Manager, Scheduler, Memory Manager, etc.).

Scheduling

ORCOS realizes scheduling on thread level. The scheduling module is distributed into the dispatcher and the scheduler.

The dispatcher is the component that assigns the thread to be executed to the processor. By the way, the dispatcher is responsible for setting and/or restoring the context of the dedicated thread.

The scheduler is the component that decides which thread is to be executed next. Therefore, it implements the scheduling strategy in terms of an algorithm. ORCOS provides several scheduling algorithms to be selected during compile time configuration. The following list contains the available real-time as well as non real-time scheduling algorithms, but as ORCOS is continuously under development, this list is not limited to these algorithms:

- Round Robin
- Priority Scheduling
- Rate Monotonic (RM)
- Earliest Deadline First (EDF)
- Earliest Deadline First with Total Bandwidth Server (TBS) for aperiodic thread scheduling
- Rate Monotonic with Background Scheduling (for dynamic task loading, see section 8.1.2)
- Earliest Deadline First with Background Scheduling (for dynamic task loading, see section 8.1.2)

ORCOS allows the execution of real-time tasks as well as non real-time tasks. If tasks with real-time constraints are loaded on the system, a specific real-time compliant scheduler is absolutely required. On the other hand, if a real-time scheduler is configured, only tasks that are derived from the `RealTimeThread` are accepted. This interdependency is ensured by specifications in the SCL-Dependencies.

6.3.2 Memory Design and Management

The Memory Management module belongs to the modules that are always needed and therefore, has always to be present in the operating system. The ORCOS memory concept is separating the kernel's and every user task's memory space from each other. ORCOS even uses virtual memory management as an additional security feature if the underlying platform supports it.

ORCOS allows some memory layout settings to be configured by SCL for the kernel as well as for each individual tasks. However, the memory layout always needs to follow the following rules:

1. Interrupt Handlers and Kernel Code are mapped one to one at the beginning of the global memory.

2. Kernel data with the kernel heap and its stack immediately follows the kernel code

3. Tasks can be placed at the remaining memory regions but must not overlap each other.

Fig. 6.3 exemplary illustrates the ORCOS memory layout for a system with two tasks. Due to the strict separation of memory spaces, kernel and tasks need their own stacks as well as heaps to operate on. The memory structure of a task is depicted in the Fig. 6.3. At the beginning of memory space assigned to a task, the task's *TaskTable* is located, followed by its code. The remaining part of the task's memory is reserved for the stacks and heaps of the threads that belong to that task. Usually the stack is filled up from the top to the bottom of the assigned memory space while the heap allocates memory upwards starting at the bottom.

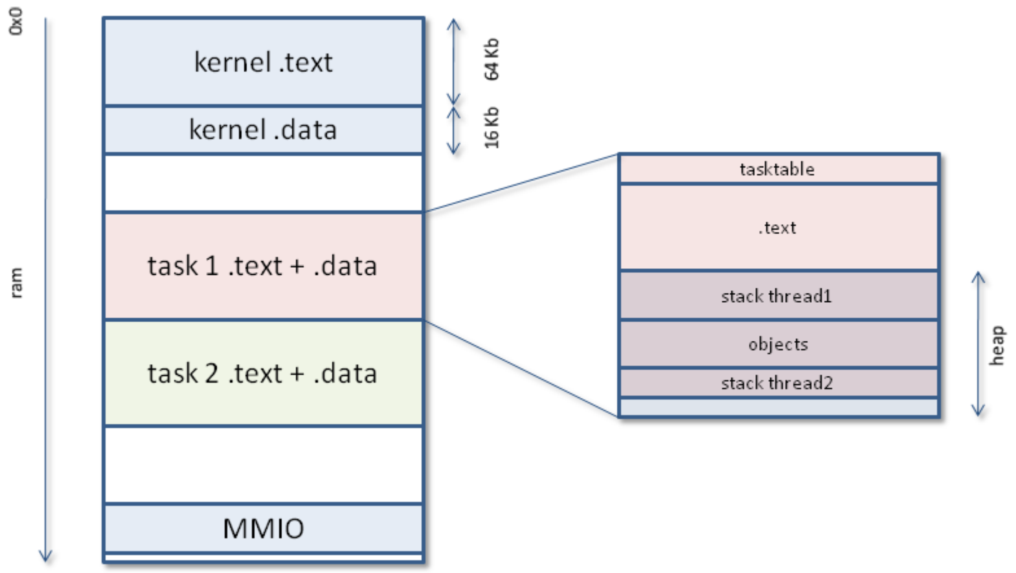


Figure 6.3: ORCOS Memory Layout; Source: [2]

Memory Management Strategies

The separation of memory requires individual memory managers for the separate memory spaces. Hence, one memory manager is responsible only for the kernel memory as well as each task owns its own memory manager. The memory management strategies are individually configurable.

Up to now, ORCOS supports two different memory management strategies:

- Linear Memory Manager
- Sequential Fit Memory Manager

The strategy implemented by the *Linear Memory Manager* is really simple. It allocates memory segments in a consecutive manner, one segment after the other. Memory that has been allocated once cannot be released (it does not implement any `free`-method). The *Linear Memory Manager* may be considered as inefficient in terms of memory usage. However, the memory allocation procedure prevents searching for free memory segments and thus, it can guarantee constant time for memory allocation. In the context of real-time systems, this property is essential in terms of time-predictability.

The *Sequential Fit Memory Manager* supports the allocation of memory as well as releasing memory. In the *Sequential Fit Memory Manager*, the entire memory is managed in a linked list of memory chunks. Every memory chunk contains a header with management information, such as the preceding and the following chunk, the payload size of a chunk as well as the state of a memory chunk. The state of a chunk defines whether the chunk is free and available to be allocated or is occupied due to allocation. There is no other structure to keep track of free memory chunks. In order to allocate requested memory, the *Sequential Fit Memory Manager* searches for contiguous memory. For this purpose, it traverses the list while searching for a free chunk of sufficient size. In case it finds more than one free chunk, the *Sequential Fit Memory Manager* provides a placement policy to decide which chunk to choose. The following placement policies are available for the *Sequential Fit Memory Manager* and, obviously, configurable by SCL:

- ***First-Fit:*** The memory manager chooses the first chunk matching the requested memory size. This placement policy is set by default.
- ***Next-Fit:*** Like in *First-Fit*, the memory manager chooses the first chunk that fulfills the condition. However, the search does not start at the beginning of the linked list, but rather at the last chunk that has been allocated.
- ***Best-Fit:*** In this placement policy, among the chunks that fulfill the conditions, the chunk is selected that has the smallest unused rest payload.
- ***Worst-Fit:*** In this placement policy, the chunk is selected that has the biggest unused rest payload.

6.3.3 System Call Manager

The ORCOS hybrid kernel design does not allow any direct communication between the tasks and the kernel. ORCOS provides system calls as the only interface between tasks and kernel functions but also for the access to hardware devices and communication. A complete list of the system calls supported by ORCOS up to now can be found in Appendix A.

The system calls invoked by user threads are handled by the `System Call Manager`. In fact, when a thread calls a system call, the kernel will be triggered using an interrupt. The interrupt handler is then responsible for forwarding the handling of the system call to the `System Call Manager` which is executing the appropriate kernel functions. Fig. 6.4 illustrates this process of system call handling in ORCOS. Handling system calls by usage of interrupts involves switching the context between thread's user mode and kernel mode.

6.3.4 Filesystem and Devices

The ORCOS filesystem is inspiredly the concept of the UNIX filesystem. All devices are registered at the Filemanager and are accessed by a unique path. By using the file paths to access system resources, the filesystem builds the interface between the kernel and the device drivers. When using a device by opening its file, the access to this resource is exclusive. All devices are automatically secured by a mutex.

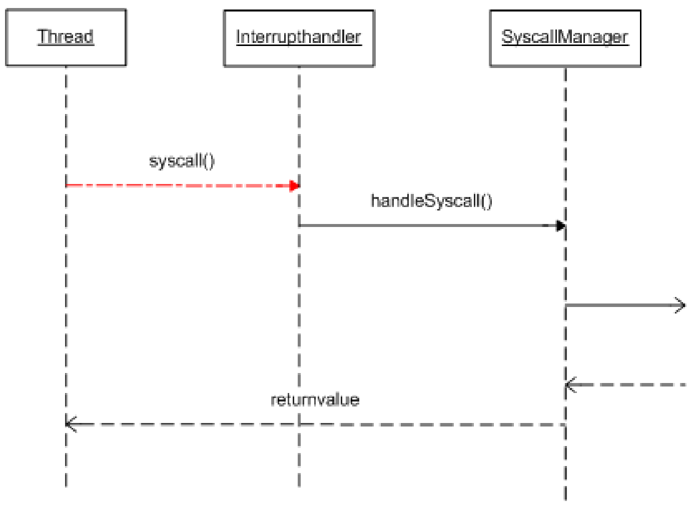


Figure 6.4: Processing of a System Call in ORCOS; Source: [2]

User tasks interact with devices through system calls by using the file paths provided by the Filesystem while the device drivers interact with the kernel via interrupts. Device drivers may be part of the kernel code as well as they may be executed in user mode. Separating the device drivers from the kernel space has great effect on the system's security as it prevents kernel code and memory being compromised by device drivers.

6.3.5 Communication

The communication module is applied for external communication of the tasks or the kernel. Compliant with the entire ORCOS design, the communication module uses a very adaptable socket interface that is completely configurable. This design makes it possible to communicate with any other system using the same interface protocol so that the communication will become transparent from the ORCOS point of view.

6.4 Summary

ORCOS is a fully offline configurable real-time operating system. As developed for embedded systems, its main objective is to address the restrictions associated with embedded platforms. The customization of the kernel allows to deliver a small footprint of the executable binary by providing full operating system functionality in terms of the configured modules required by the applications. Because of its flexibility, ORCOS offers an adequate platform for research evaluations. Additionally, ORCOS was extended towards online reconfigurability and implementing self-x properties (see chapter 8) which build up the basis for evaluating this Online Anomaly Detection approach.

ORCOS implements security features as it strongly separates the operating system kernel from the applications by providing system calls for interaction between the kernel and the applications. As system calls are the only interface between applications and the operating system, this strong separation ensures to prevent any intended or non-intended interference or damage of the kernel structures from the application side. Additionally, this clear separation builds the best prerequisite for clearly defining and extracting application behavior dealing as input for the anomaly detection.

The implementation of the operating system functionality in forms of kernel modules makes the modules exchangable on the one hand. On the other hand, the modules are also clearly separated from each other which again offers best prerequisites for evaluating the operating system health state (required for the Online Anomaly Detection) based on the enclosed small components.

Because providing all required prerequisites for implementation and evaluation, ORCOS will be applied for integrating the Online Anomaly Detection approach proposed in this thesis.

Part III

Online Anomaly Detection

CHAPTER 7

Online Anomaly Detection for Reconfigurable Real-Time Systems

The objective of this thesis is to enhance the run-time dependability of self-reconfiguring real-time systems by introducing Anomaly Detection. The combination of real-time systems and self-x capabilities makes the integration of existing approaches impossible as the specific characteristics of those systems result in more challenging requirements. In contrast to many existing anomaly detection approaches which have been analyzed in Chapter 4, in dynamically changing systems it is not sufficient to rely on the principle of classifying execution data based on the fact whether they are known or unknown to the anomaly detection system. Another principle - using different classification criteria - has to be applied to enable the evaluation of system behavior in self-x environments. Referring to the challenges described in Chapter 1.1, the approach has to be able to cope with novel and previously unknown behavior and, therefore, be self-learning. The analysis of the according execution data is required to be lightweight with low memory consumption and deterministic computation time, as it has to be performed online because of the criticality of the application domain. This chapter presents the concept of the Online Anomaly Detection designed for self-x real-time operating systems.

The development of the approach primarily required the determination of an adequate method to evaluate and classify system behavior. The Danger Theory has been identified as an appropriate source of inspiration as it provides a different classification perspective: instead of examining behavior data with respect to a predefined knowledge, anomaly detection performed on the basis of the Danger Theory relies on the examination of the environmental context of collected behavior data. The environmental context is defined to be represented by (absence or presence of) danger signals reflecting the threat potential caused by the behavior. Hence, this method allows to examine known as well as unknown behaviors by considering their threat potential to the system.

In order to appropriately integrate the Danger Theory into a real-time operating system, it is necessary to analyze the specific execution data generated by the real-time operating system, its components and its application. The execution data serves as the input data of the anomaly detection. As a basis, in the first part of Section 7.1, the concrete data of interest was defined by considering the available execution data in real-time operating systems, divided into data that specifies behavior and data that determines the contextual threat potential. Based on this definition, the required components could be identified that establish the Anomaly Detection Framework that is presented in Section 7.2. We have already published the ideas for the Anomaly Detection Framework in [Stahl et al., 2013, Stahl, 2013, Stahl et al., 2014b]; the framework enables a context-related classification of behavior represented by system call sequences on the basis of input signals reflecting the operating system state.

In order to specify the classification process implemented by the Anomaly Detection Framework, a detailed analysis of the anomaly detection problem and its required features was required. To ensure a formal specification of the anomaly detection problem, we have categorized the targeted anomaly detection type and the expected output data according to the classification of Anomaly Detection Systems presented in Chapter 2.2.1. The features and challenges required to be realized by the classification method have been selected by considering the specific characteristics inherent to the application domain. To ensure a systematic procedure and completeness, the feature selection process was performed in accordance with the design decision process presented in Chapter 2.2.2. The selected features presented in Section 7.1.3 had impact of the design on the Anomaly Detection Framework and, in particular, on the introduction of the Behavior Knowledge Base that is presented in Section 7.3. As already published in [Stahl and Rammig, 2014, Stahl and Rammig, 2015], the Behavior Knowledge Base based on Suffix Trees is responsible for online behavior profiling as well as for keeping the history that builds up the basis for the learning capability of the approach e.g. for the purpose of enhancing the classification of recurring behaviors.

This chapter presents the core approach of the Online Anomaly Detection consisting of two main ingredients: the Anomaly Detection Framework inspired by the Danger Theory to enable context-related classification of system behavior, and the Behavior Knowledge Base to profile behavior and memorize the classification outcomes to ensure self-learning. Considering these two ingredients, an architectural model for the approach implemented was derived on the basis of the architectural model presented in Chapter 2.3 illustrating the data flow of the Online Anomaly Detection. The resulting architectural model completes the concept and is presented in the final section of this chapter, Section 7.4.

7.1 Problem Definition and Feature Requirements

The particular problem addressed by this thesis is the Anomaly Detection in self-x real-time operating systems. Real-time systems posses specific characteristics that have to be included into the concept of the anomaly detection. As system behavior is dedicated to be evaluated by the anomaly detection approach, the definition of behavior is essential to determine the data of interest as required by any anomaly detection system. Section 7.1.1 provides this definition. Based on this, the anomaly detection problem is classified in section 7.1.2 leading to the specification of design decisions and features in section 7.1.3.

7.1.1 Defining Behavior

Operating systems are service platforms for the applications responsible for managing their execution and providing access to the available resources in a stable and dependable manner. They offer interfaces to their services, usually by specifying system calls that implement the functionality at operating system side. Such interfaces prevent direct access to operating system resources and enable protection of system resources as well as the kernel. Applications can only interact with the operating system and gain access to the present resources by using the interface and, in particular, the system calls. Executing a system call internally changes attributes of resources and/or the kernel, of course, with respect to the executed system call and its arguments. The attributes make up the state of the operating system which is composed of global operating system parameters such as CPU usage and kernel parameter values, but also parameters associated with the available operating system components such as state of a resource, resource consumption, memory usage, computation load, etc. In accordance with the present concept, we call the operating system state the *Health State* H. The *Health State* is related to a particular point of time t defined as:

$$H(t) = \begin{pmatrix} p_1 \\ p_2 \\ \dots \\ p_m \end{pmatrix}$$

Any p_i represents a relevant operating system (component) parameter that contributes to the state. The size of the *Health State* vector is related to the number of parameters that the operating system defines. Hence, it is specific to the operating system implementation including its integrated components.

Because of a clearly defined and protected interface, an application can only impact on the internal operating system state by using the system call interface. Hence, the application behavior mainly effects the system's *Health State*. (Any other application activity can have effect on the application's internal structures, attributes etc. but without any contribution to the operating system state.)

System calls being the interface between operating system and application, we define the behavior of applications by their system call invocations and their associated information (arguments, return values, return addresses etc.). In this thesis, as a first step we only concentrate on the pure system calls without considering the associated information. Therefore, *Behavior* S_i of a task i is composed of the sequence of system call s_k, with s_k being the identifier of the system call:

$$S_i = (s_1, s_2, \dots, s_m, \dots)$$

The *Behavior* S_i forms a time series $s_1, s_2, \dots, s_m, \dots$ representing the timely ordering of the system call invocations made by the application task i. Furthermore, applications in real-time systems in most cases are implemented as periodic tasks executing one task instance per period. The task instance terminates within is associated period. Hence, we assume the *Behavior* S_i of a real-time task to be enclosed by its period and associated with a particular instance j. The execution sequence of a task instance is of finite length n_j, specific to the task instance's executing control flow, resulting in S_{i_j}:

$$S_{i_j} = (s_1, s_2, \dots, s_{n_j})$$

Such a behavior sequence S_{i_j} is produced in each execution period of a task and can be extracted within the operating system without any application-specific knowledge or access to the application's internal structures. Furthermore, as each system call may alter the operating system state, the *Health State* vector $H(t_{s_k})$ can be determined after each system call execution (with t_{s_k} the termination time of system call s_k).

7.1.2 Problem Classification

After having defined the data of interest in terms of behavior and system state, a classification of the anomaly detection problem is required according to the foundations of Anomaly Detection as introduced in Chapter 2.

The behavior is specified as the sequence of system calls executed by an application task. The overall behavior of tasks is composed of the set of individual system call sequences of the tasks executing on the system. A system call sequence, represented by the IDs of the system calls, consists of discrete values with a value range determined by the set of system calls that the operating system offers.

To determine an adequate method for anomaly detection, the scope of the anomaly detection has to be identified considering the application context and the data of interest defined as a basis. This question is related to the type of anomaly addressed by the method as well as a measure for the differentiation of data items to achieve an adequate output:

1. **Anomaly Type**:

   The problem of anomaly detection for self-x systems has been motivated in Chapter 1. Furthermore, when analyzing the applicability of existing approaches (see Chapter 4) to our application purpose, it turned out that analyzing system calls or their sequences based on classifying them into known or unknown ones is not sufficient in self-x environments. It is essential to take into account the effects of the executing system calls on the system state. This means that the system call-based anomaly detection in self-x systems must be context-related, yielding to a *Contextual Anomaly Detection* approach.

   System call sequences form sequential input data for the anomaly detection. We assume that, on the one hand, a single system call is able to harm the system by e.g. provoking an exceeding of resources or by causing a failure due to incorrect arguments, etc. On the other hand, a particular sequence of system calls (referred to as *pattern* in the following) may lead to instabilities or system failures, e.g. patterns that break the dependencies between system calls (e.g. releasing resources before they are allocated) or such ones that because of their successive combination cause failures in components where each single system call is harmless. The patterns can be identified as entire behavior sequences as well as subsequences of behavior sequences.

   Hence, in our approach, we address both: the detection of *Point Anomalies* as well as *Collective Anomalies*.

2. **Labels & Output**:

   Usually, anomaly detection systems differentiate between *normal* and *anomalous* data items and specify the applied labels accordingly. In our approach, we are more interested in differentiating between *safe* and *dangerous* behavior (*safe* and *dangerous* states produced by behavior) instead of between *known* and *unknown*. The output of the classification of data items has to be performed in a context-related manner by taking into account the monitored behavior and its effects on the system state.

   Furthermore, in dynamically changing systems previously known behavior sequences may alter their effects on the operating system state from being *safe* leading into a *dangerous* state at another point of time. Such changes in classification of behavior effects can be abrupt. But it is also possible that, for a particular behavior sequence, the transition from being *safe* to *dangerous* may evolve in a creeping manner, and may transit a grey zone between *safe* and *dangerous*.

   In general, we assume that there is no clear boundary between *safe* and *dangerous*. In particular, if applying approaches that incorporate a certain degree of uncertainty because of using probabilistic classification rules, there might be a range where behaviors cannot definitely identified as *safe* or *dangerous*. They may lie in between showing up potentials to suspicion. Hence, for the purpose of this approach, we define three types of classification labels to be applied:

- safe (*green*)
- suspicious (*yellow*)
- dangerous (*red*)

The labels introduced here are assigned colors according to the traffic light principle.

The anomaly detection approach addressed by this thesis can be classified as a *Contextual Anomaly Detection* problem able to identify *Point Anomalies* as well as *Collective Anomalies* in system call sequences. Its objective is to deliver an output represented by the labels *safe*, *suspicious* and *dangerous* as the classification result when analyzing the application behavior in context-related manner.

7.1.3 Feature Selection

The basic requirements resulting from the problem domain have been formulated in Chapter 1.1. Altogether, anomaly detection for self-x real- time systems has to be autonomous, lightweight and to operate in an online manner by taking into account the specifics of real-time systems and the challenges associated with self-x systems. These requirements combined with the problem definition and classification form a basis for making design decisions, as formulated in Chapter 2.2.2, and selecting the required features that have to be built into the anomaly detection approach.

Anomaly detection systems, mainly, perform the tasks of data collection and classifying it based on a predefined or trained *normal* profile. In self-x systems, the behavior is expected to change dynamically either due to step-wise evolution or because of a system reconfiguration. This raises two challenges on the anomaly detection: First, each novel behavior configuration requires an according *normal* profile. Keeping *normal* profiles for all possible behavior configurations can violate the system's resource-restrictions. Second, it is challenging or even impossible to determine all possible *normal* behaviors or system configurations as they are based on autonomous decisions that are not completely predictable. Generating a set of - obviously predictable - behaviors and configurations to train the system *normal* profile would always be considered as incomplete. In case of a change to a previously unknown behavior or configuration, it is impractical to induce a training phase due the fact that, on the one hand, the system is in operation and, on the other hand, the anomaly detection is required to continue its execution as it has to operate in an online manner. Hence, the anomaly detection is required to work **without any training phase** and consequently has to learn behavior online. This requirement makes this approach unique as the works discussed in Chapter 4 (Related Work) presuppose a training phase as a basis.

The anomaly detection approach addressed in this thesis relies only on the operation phase where behavior data has to be collected, classified and conserved in a knowledge base in order to enhance the classification of recurring behavior patterns. From this workflow, we derive two main responsibilities:

1. **Learning of Behavior** by collecting and storing it with its associated classification outcome
2. **Classification of Behavior** by applying a context-related method that incorporates the effects of the behavior on the system state

In relation to Chapter 2.2.2, the design decisions have been specified for this approach with respect to the twofold responsibility of the anomaly detection:

- **Detection principle: Programmed vs. Self-Learning**:

  The overall approach is required to be self-learning. However, considering the responsibilities separately, the situation is diverse:

  In particular, without any training data set, the behavior knowledge has to be built up on observation data as the only basis. This makes the approach to become fully self-learning in terms of learning of behavior.

  This is different for the classification of behavior: The system state plays an important role for the classification outcome of an evaluated behavior sequence. The classification can rely on thresholds and/or conditions that are formulated by means of classification rules. On the one hand, rules specified for classification of behavior can be programmed by making use of expert knowledge. E.g. a dangerous system state is present if the usage of a particular resource exceeds the available amount. Intuitively, this can be specified by threshold-based rules. On the other hand, the classification rules can implement self-learning capabilities. This can be useful, for example, if the progress of state parameters is of interest to assess the system state.

- **Mode of Anomaly Detection Technique: supervised, semi-supervised or unsupervised**:

  In terms of determining the detection mode of the desired approach, again, we have to split the discussion into considering the responsibilities of behavior learning and classification in isolation:

  The process of learning behavior is completely unsupervised and only ruled by the fact that behavior is defined by system call executions of tasks. Any occurring behavior has to be monitored and collected, primarily without considering its classification.

  In contrary, the classification method is required to deliver a classification outcome. Diverse methods for assessing the system state are possible so that the classification rules applied can be supervised, semi-supervised and/or unsupervised.

- **Time of detection**:

  In relation to the basic requirement, online anomaly detection is characterized by the ability to identify threats at the point of time they occur. Applied for real-time systems, this is essential in order to prevent a propagation of a failure throughout the system with potentials to result in critical consequences for the system, its applications or even humans involved. Hence, the time of detection is defined as *real-time*, but in contrary to Axelsson [23], we prefer the term *online* here.

- **Granularity of data-processing**:

  The granularity of data collection is determined by the definition of behavior. System calls are monitored on demand when being executed which is classified as *online* and *continuously*.

  As any system call has the potential to harm the system, each system call execution has to be evaluated immediately. In order to enable an online classification, the classification of each executing system call must be performed *online* to ensure a *continuous* classification.

  Storing the occurring behavior sequences, for example within a knowledge base, preserves the history of execution of a task and is essential for ensuring learning. Information such as which sequences occur predominantly often may be helpful for establishing a kind of *normal* profile. This requires counting the number of occurrences of a sequence. In addition, recording the classification outcome of a sequence enables to preserve the progress of a particular sequence in terms of effects on the system state. In order to realize a knowledge base that holds all these information, a pattern matching must be implemented to enable the identification of equal sequences. Thereby, it is essential that the pattern matching method is designed for **exact matching** of patterns as two sequences that differ even by only one symbol in the sequence (one system call) may achieve totally different classification results.

- **Locus of data-collection**:

  The behavior of each task is individual and, therefore, each task is required to have assigned its own data collection and storage. Considering each task as a data source, data-collection is realized in a *distributed* manner as multiple data sources exist. Behavior data in forms of system call sequences (by IDs) can be *directly* extracted by the operating system without any need of being preprocessed.

  For the system state parameters dealing as input data source for classification, the situations is not that clear. System state parameters can be distributed throughout the individual operating system components, building up *distributed* data sources. However, the operating system contains also global system parameters that are held in a *centralized* manner. The diversity of state parameters does not allow for an explicit categorization into *direct* or *indirect* data collection as the state parameters have not been defined at this stage and are specific for each operating system implementation. We assume that there are parameters that can be directly extracted from the operating system, e.g. currently allocated memory size. But parameters also exist that have to be preprocessed such as an utilization factor of a resource. Therefore, the data collection of state parameters will partly be performed in *direct* as well as in *indirect* manner depending on the data type and the operating system implementation.

- **Locus of data-processing**:

  Even though the input data is collected in a distributed manner, the classification rules applied shall be uniform for all tasks and, hence, for all behavior sequences. Therefore, in order to ensure a consistent classification of behavior data, the data-processing has to be performed in *centralized* manner.

  All behavior data as well as system state data are available in the operating system. In order to guarantee *online* detection, the anomaly detection has to become an inherent part of the operating system leading to an *internal* anomaly detection approach.

- **Response to detected anomalies**:

  Addressing the application domain of real-time systems, an identified anomaly representing a potential system threat has to be immediately signaled and transferred to the proper authority responsible for reaction. Hence, the response to detected anomalies is classified as *active*.

  In this thesis, we are only interested in the detection of suspicious or dangerous behaviors in order to send according alerts. The reaction on identified threats can be diverse: stopping a task execution and eliminating it from the system, recovering back to a dedicated safe state, restarting, reconfiguring application or operating system parts, etc. The decision how to react is not straightforward as it may have consequences on the system performance. While for one application a reset might be sufficient, it could harm the overall service of the system for another one. In particular, in order to generate an adequate response or reaction on an unsafe behavior, threat-specific and application-specific knowledge is essentially required.

- **Ability to evolve**:
  Based on the requirement of the approach to be self-learning, the ability to evolve is inherently included.

The concept for anomaly detection has been developed by taking into account all these requirements described above. Of course, any anomaly detection approach is assessed by its coverage and accuracy which we address to achieve by the concept.

7.2 Anomaly Detection Framework

Considering the requirements, the biggest challenge is to find an approach that can work without any training phase allowing for a classification in context-related manner. The Danger Theory from Artificial Immune Systems (see Chapter 3.4) provides a good source of inspiration for the addressed problem: it allows assessment of observed behavior by considering its current context represented by presence or absence of so called *danger signals* without any need of matching it against a trained *normal profile*. Dendritic Cells (DC) build up the core of the population-based approach responsible for monitoring the system entities, they are assigned to, and performing - on the basis of provided input signal - a state transition that forms the classification method of the DC.

Transferring this concept into our problem domain, DCs are responsible for observing the system behavior defined by the system call sequences of tasks. Hence, a DC is assigned to each task instance responsible for collecting data in forms of executed system calls. For state transition, a DC requires input signals that in our case is the system state defined as the *Health State* delivered by the operating system.

7.2.1 Operating System Architecture

For the anomaly detection to become an internal part of the operating system, the operating system architecture has to be extended by introducing required components. Fig. 7.1 illustrates the resulting Anomaly Detection Framework consisting of DCs, each associated to one task (the current instance), a Health State Monitor for delivering the input signals, a classification module responsible implementing the conditions for state transition of the DC and a Knowledge Base for conserving the migrated DCs.

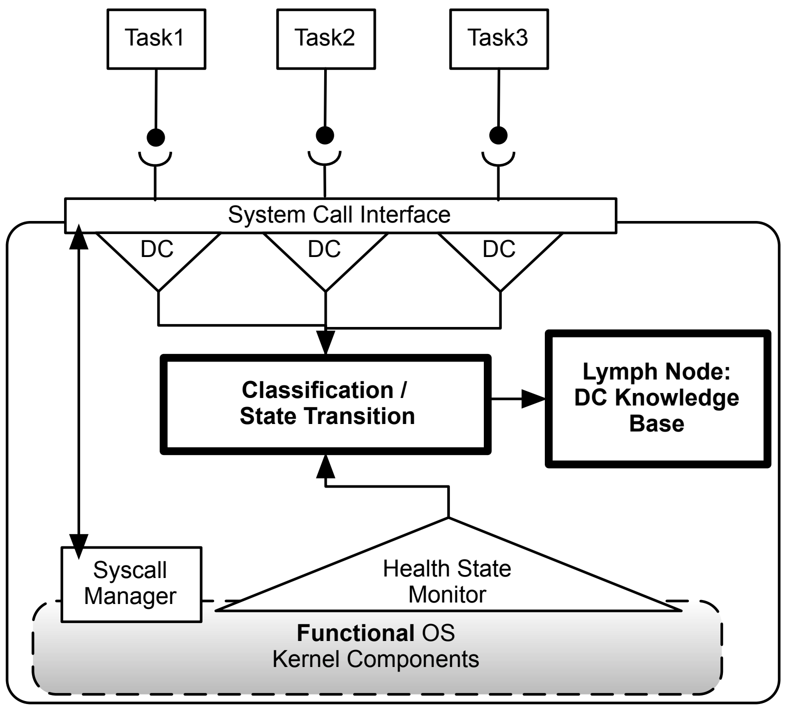

Task1
Task2
Task3
System Call Interface
DC
DC
DC
Classification / State Transition
Lymph Node: DC Knowledge Base
Syscall Manager
Health State Monitor
Functional OS Kernel Components

Figure 7.1: Anomaly Detection Framework integrated into an Operating system.

7.2.2 Classification Input Signals

The behavior data collected by the DC has to be evaluated based on the present signals reflecting the *Health State*. The Danger Theory defines four input signals that in context of the operating system *Health State* are specified as:

Safe signal:
indicates that no threat has been identified in the system; the operating system state is in *normal* or *safe* operation with respect to its system state parameter values.

PAMP (pathogen-associated molecular pattern) signal:
indicator that a known threat has been localized; the PAMP signals occurs in case of a component failure or state parameter values that signal a definite malicious operating system state

Danger signal:
indicates a potential danger suspected; the operating system state parameters are not in the *normal* or *safe* operation range, they show up an anomaly (but not a known threat such as indicated by the PAMP signal)

Inflammation signal:
general alarm signal; the inflammation signal is applied to signal a reconfiguration in order to indicate a potential change in system behavior or system state

The particular specification of the *Health State* can only be conducted on the basis of the present operating system parameters and is strongly dependent on the operating system implementation. However, as the Health State Monitor is responsible for detecting anomalies in the operating system state, it can be considered as an anomaly detection system itself that works internally.

7.2.3 Behavior Classification

The lifecycle of a DC is determined by the lifecycle of the task instance that the DC is responsible for. Any system call executed by that task is collected by the DC and evaluated in the context of the input signals. The idea of the classification method is simple: if the resulting *Heath State* after a system call execution is reflected by a *safe signal*, the classification outcome of the system call is a *green* signal. If the *Health State* caused by a system call execution is a *danger signal*, the classification outcome is *yellow* representing a suspicion. If the *Heath State* delivers a *PAMP signal*, the execution of the system call is classified as *red* indicating a system danger.

Formally integrating the classification into the concept of Dendritic Cells, it is performed at the state transition of the DC (see Chapter 3.4.2). Three state are defined for Dendritic Cells: *immature*, *mature* and *semi-mature*. The initial state of the DC is *immature*. Two exclusive conditions can initiate a state transition:

1. The lifecycle of the DC expires. This happens when the tasks instance terminates.

   As long as the task instance is active, the DC continuously collects system calls and evaluates them immediately based on the present *Health State* signals. It remains in *immature* state as long as no *PAMP signal* occurs, - irrespectively whether the input signal is *safe* or *danger* as this only indicates a suspicious but not a true danger.

   When the task instance terminates, the state transition of the DC is enforced: In presence of *safe signals*, the DC migrates into *semi-mature* state and outputs a *green* classification outcome assigned to the task instance's behavior sequence. In case a *danger signal* occurred during the system call execution of the task instance, the DC migrates to the *mature* state with the *yellow* signal as classification outcome.

2. The presence of a *PAMP signal*. If the *PAMP signal* occurs during the task instance's execution, it signals the presence of a true threat. Then, the task's execution is immediately aborted in order to prevent further damage caused by continuing the execution. The DC directly migrates into the *mature* state and assigns a *red* classification outcome to the behavior sequence executes until then.

By this method, the behavior data is directly brought in context to the system state parameters. This method allows to classify any occurring behavior which provides a good basis for coverage. The fact that any behavior is evaluated individually, in turn, ensures a certain degree of accuracy but this one is strongly dependent on the accuracy of the *Health State* evaluation. Implemented within a classification component, this lightweight principle of data-processing allows the classification to be performed online and directly inside the operating system.

7.3 Behavior Profiling and Knowledge Base

In the Danger Theory, Dendritic Cells migrate to the lymph node after their state transition in order to be memorized to enhance future detection and reaction. The same reasons motivate to introduce a storage for behavior sequences:

1. conserving the history of execution enables to determine a kind of *normal profile* established on the basis of a behavior sequences with a large number of occurrence. In fact, real-time applications are expected to exhibit *similar* behavior sequences, at least with common subsequences, throughout their task instances. Knowing whether a behavior sequence is commonly executed or rather rarely makes the classification results more reliable, in particular for commonly executed behavior sequences belonging to the *normal profile*.

2. the previous classification outcome(s) of a behavior sequence, and even of each single system call within that sequences, may be of interest to enhance the classification of a current occurrence of that behavior sequence. The classification outcome of a previous execution provides a tendency of the effects of that particular behavior sequence. Conservation of this information allows to monitor the progress of effects of a particular behavior sequence on the system state. Furthermore, knowing that a current behavior sequence matches a pattern that has previously been determined as *dangerous* could prevent damage on the system by exploiting the classification information and aborting the execution of the behavior sequence at the earliest possible stage.

Storing behavior sequences with their associated classification outcomes in a Behavior Knowledge Base forms the basic instrument of learning. With respect to the application domain, several requirements are addressed on such a Behavior Knowledge Base as described above: Due to the restrictions in terms of memory space, the Behavior Knowledge Base is required to store behavior sequences in a compact manner. Matching, in particular exact matching, of sequences has to be performed online, in a symbol-by-symbol manner synchronously to the system call invocations with low computational effort. Furthermore, the Behavior Knowledge Base must be extendable by novel arising behavior sequences in order to ensure the learning process.

Considering the specifics of behavior sequences being of finite length, and expected to be rather similar throughout the task instances, Suffix Trees - as introduced in Chapter 5 - prove to be attractive for this application purpose: Suffix Trees provide a symbol-by-symbol construction method for sequences introduced by Ukkonnen that allows to include the system calls into the Suffix Tree as soon as they occur without the sequence being completed. Furthermore, Suffix Trees offer a successive online matching method in linear time (linear to the length of the sequence). Even though, the memory consumption for a single behavior sequence of length n is $O(n^2)$, it is overall acceptable because it contains all sub-sequences of the sequence and consumes no additional memory space in case of recurring sequences (or their subsequences).

Based on the fact that a Suffix Tree in the Behavior Knowledge Base is intended to store multiple behavior sequences of a task - or moreover, its complete behavior sequence history - being more accurate concerning the terminology, in fact, *Generalized Suffix Trees* are addressed here.

7.3.1 Behavior Knowledge Base Properties

The concept of Suffix Trees is required to be modified for the application purpose: First, we are not interested in starting positions of suffixes originally stored in the leaf nodes of the Suffix Tree. They become obsolete as the Suffix Tree is intended to store multiple sequences that potentially share the same paths. Second, the data structure of the Suffix Tree has to be extended by additional information required by the approach: a counter of occurrences of

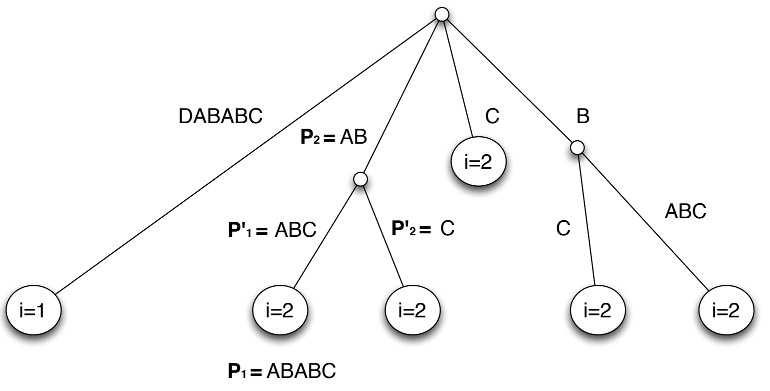


Figure 7.2: Behavior Knowledge Base with Occurrence Counter in the Suffix Tree containing the example sequences $S_{i1} = ABABC$ and $S_{i2} = DABABC$.

behavior sequences and a marker to store classification outcome assigned to each sequence symbol.

1. **Occurrence Counter**:

   Each leaf node is extended by a *Occurrence Counter*. Whenever a system call sequence is executed, the *Occurrence Counter* of the leaf node identifying the matching path is incremented. To determine the number of executions of sub-sequences, we can exploit a property provided by Suffix Trees: originally, counting the number of occurrence of a sub-pattern within a sequence is simply done by counting the number of leaves that arise from the (inner) node that determines the sub-pattern. According to this property, the number of occurrence of a sub-sequence ending in an (inner) node is determined by the sum of the *Occurrence Counters* of all the leaf nodes that arise from that shared node.

   For better understanding we provide an example illustrated in Fig. 7.2. It shows the Behavior Knowledge Base Suffix Tree for sequences $S_{i1} = ABABC$ and $S_{i2} = DABABC$ being executed. The pattern $P_1 = ABABC$ is a subsequence of S_{i1} and S_{i2}, as S_{i1} is a suffix of S_{i2}. When counting the occurrence of P_1, it is 2 as it is executed once in S_{i1} and once in S_{i2}. In the Suffix Tree illustrated in Figure 7.2, the *Occurrence Counter* of P_1 is set to $i = 2$. To calculate the occurrence of the subsequence $P_2 = AB$, the property of the Suffix Tree is exploited: adding the *Occurrence Counters* of the descending paths $P'_1 = ABC$ being $i = 2$ and $P'_2 = C$ being $i = 2$, the resulting *Occurrence Counter* of subsequence P_2 is $2 + 2 = 4$.

   Integrating a *Occurrence Counter* into the leaves of our Suffix Tree makes it possible to calculate the number of occurrences of all (sub-)sequences that are contained in the Behavior Knowledge Base.

2. **Classification Marker**:

   Each system call is intended to be classified individually. To preserve the classification outcome, the Suffix Tree is extended by an additional attribute assigned to the suffix tree labels: the *Classification Marker*. According to the defined classification labels, the *Classification Marker* colors the suffix tree labels in *green*, *yellow* or *red* to reflect the result of the classification. Finally, as the path of a behavior sequence in the Suffix Tree is unique and ending in a leaf node, the leaf nodes are equipped with a *Classification Marker* as well to represent the classification outcome associated with the entire behavior sequence.

By introducing the *Classification Marker* into the Suffix Tree, the Behavior Knowledge Base can preserve all behavior sequences in one data structure but differentiate between *safe*, *suspicious* and *dangerous* behavior based on the value of the *Classification Marker*.

Profiling the behavior online requires every executed system call to be processed on demand. The system call processing defines a workflow composed of *Extending*, *Classifying* and *Updating*. *Extending* the Suffix Tree by the system call first involves verifying whether an entry for that system call already exists in the path of the current sequence. If no matching in the current path label is found, the Suffix Tree has to be extended according to the specifications of Suffix Trees. In case of a match, this step is skipped. *Classifying* a system call is concerned with requesting the *Health State* after its execution, while *Updating* includes setting the resulting value to the associated *Classification Marker* and incrementing the *Occurrence Counter*, in case a leaf node is reached.

At the initialization of the system, the Behavior Knowledge Base is empty. With the execution of the first instance of the task, the Suffix Tree is constructed by applying the basic symbol-by-symbol online construction method specified by Ukkonnen. The Behavior Knowledge Base is extended by any behavior sequence executed by a task and it stores its classification labels which allows continuous learning of behavior and updating of classification results.

In particular, adaptations and reconfigurations induce the emergence of novel behaviors. The extendability of the Behavior Knowledge Base enables the approach to become applicable for the purpose of reconfiguration.

7.4 Architectural Model

The anomaly detection approach relies on classifying behavior data on the basis of operating system health state information. Data processing is performed on two levels: behavior data as well as operating system health state data. Classification input signals, in particular the *danger signal* and the *PAMP signal*, are raised in case of anomalies in the operating system state parameters. Moreover, the accuracy of the behavior classification is determined by the accuracy of the health state evaluation. Anomaly detection, in fact, is performed on two levels: anomaly detection concerning behavior and the evaluation of the operating system state that is becoming an internal anomaly detection itself. To determine the concerned system components and their responsibilities, we specified an architectural model on the basis of the model defined in Chapter 2.3 and determined the data flow within the architecture. As coping with an anomaly detection on two levels, the model was adjusted accordingly and consists of an Application Behavior Anomaly Detection and a Health State Anomaly Detection as shown in Figure 7.3.

On the Application Behavior Anomaly Detection level, the DC needs to collect behavior data and evaluate it. For the DC, the system call data is extracted by a System Call Monitor and directly included into the Suffix Tree Behavior Knowledge Base. After storing, its processing is performed by Classification that requires input data from the Heath State Anomaly Detection level. The Application Behavior Anomaly Detection delivers a Classification Outcome as the anomaly detection alarm.

The Health State Anomaly Detection relies on data collected by an OS Health Monitor. This data is stored in a OS Health Monitor Database that deals as input source for the Signal Generator responsible to deliver the Health Signal of the system. The Signal Generator operates

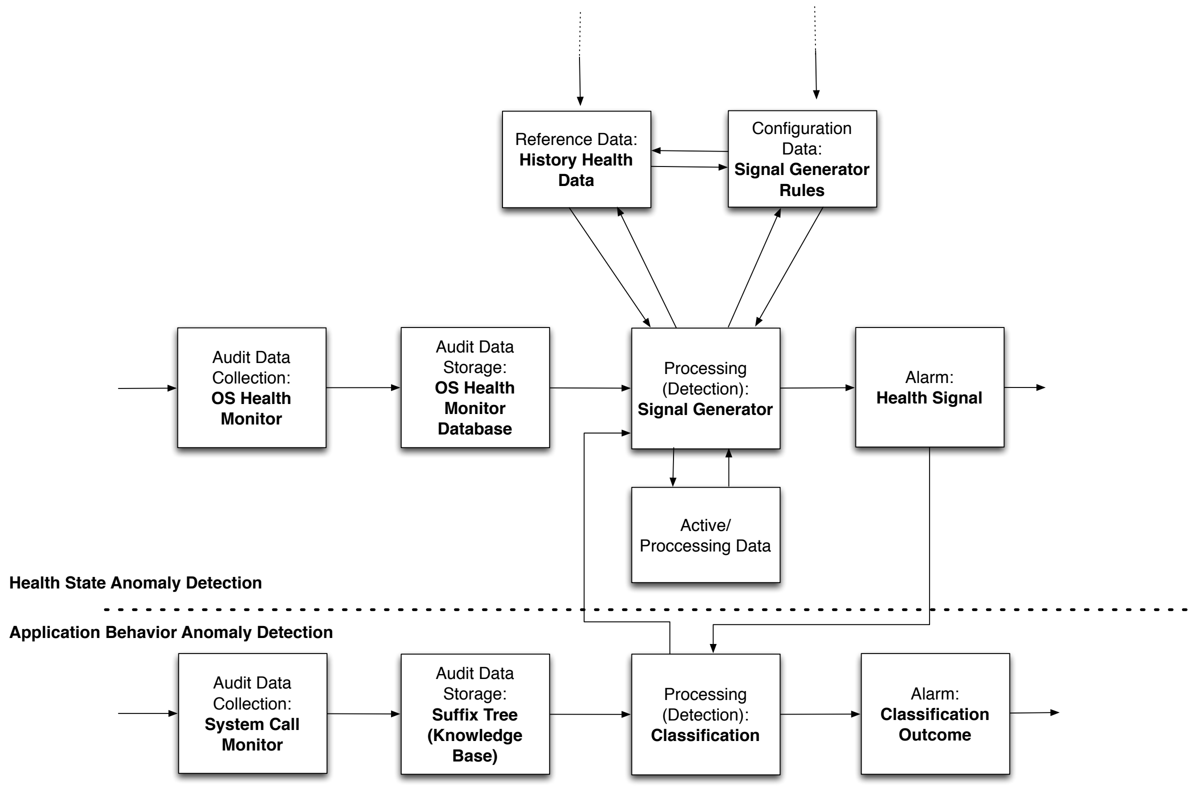

Reference Data: History Health Data
Configuration Data: Signal Generator Rules
Audit Data Collection: OS Health Monitor
Audit Data Storage: OS Health Monitor Database
Processing (Detection): Signal Generator
Alarm: Health Signal
Active/ Proccessing Data
Health State Anomaly Detection
Application Behavior Anomaly Detection
Audit Data Collection: System Call Monitor
Audit Data Storage: Suffix Tree (Knowledge Base)
Processing (Detection): Classification
Alarm: Classification Outcome

Figure 7.3: Architectural model for Online Anomaly Detection (referring to Chapter 2.3, Fig. 2.1)

on the data in the OS Health Monitor database, the History Health Data as reference data and Signals Generator Rules that formulate the condition rules to obtain a result. The output of the Health State Anomaly Detection is the Health Signal that is passed over to the Application Behavior Anomaly Detection level.

7.5 Summary

This chapter presents the concept of the anomaly detection approach that was developed based on the clear and detailed specification of the addressed problem application domain and the resulting requirements. System behavior is defined by system call sequences executed by task instances and has to be evaluated in order to assess the danger potential of the behavior on the system.

The core of the approach is the Anomaly Detection Framework inspired by the Danger Theory and the Behavior Knowledge Base implemented in form of Suffix Trees. The Anomaly Detection Framework requires an integration into the architecture of an operating system. For evaluation and classification of behavior sequences, the Danger Theory provides a good source of inspiration as it allows a context-related classification that takes into account the effects caused by the executed behavior sequences. The context-related classification builds the foundation for anomaly detection in self-reconfiguring real-time systems and forms the basis for an adequate accuracy. The classification of all executed behavior sequences ensures the coverage.

The Behavior Knowledge Base is responsible for conserving the behavior sequences with their classification outcomes. Suffix Trees provide best prerequisites for a data structure applied for the Behavior Knowledge Base because of the characteristics of behavior sequences obtained by executions of real-time tasks. The online-construction method of Suffix Trees allows online profiling of behavior and a continuous updating of the Behavior Knowledge Base which is also important for supporting changes in behavior caused by reconfigurations. By adjusting the concept of Suffix Trees to preserve the multiple behavior sequences, adding information such as *Occurrence Counters* and *Classification Marker*, we ensure the storing of the entire execution history.

On the basis of this concept, we have developed an Architectural Model that specifies components with responsibilities to be implemented and integrated into a concrete real-time operating system.

Part IV

Implementation and Evaluation

CHAPTER 8

ORCOS Online Anomaly Detection Framework

ORCOS (introduced in Chapter 6) is used as the evaluation platform for the Online Anomaly Detection approach presented in this thesis. The topic of this chapter is the implementation and integration of the Online Anomaly Detection approach into this operating system.

Some work was made on ORCOS in order to make this integration possible: The basic architecture provided by ORCOS was extended by a Profile Framework to enable online reconfigurability (see section 8.1) as a prerequisite for (dynamic) autonomous behavior - including self-x properties. For the integration of self-x methods in ORCOS, a generic framework was designed which allows monitoring, analyzing and controlling of the system, as described in section 8.2. In order to apply Online Anomaly Detection as a means of self-diagnosis based on system calls in context to the health state of the system, the previously proposed generic framework has been adopted to its application purpose in section 8.3. The monitoring of system calls has been implemented by a System Call Monitor specified in section 8.4. Out of this monitored data, a Knowledge Base is constructed by a representation applying Suffix Trees illustrated in section 8.5. Data in the Knowledge Base serves, on the one hand, as a storage but, on the other hand, it offers the input for the classification in the context of the system health status. The system's health data is provided by the System Health Monitor depicted in section 8.6. Section 8.8 is completing this chapter as it presents the integration of all components of the ORCOS Online Anomaly Detection Framework and shows how they interact and operate at runtime.

8.1 Online Reconfigurability

As already emphasized, ORCOS is an execution platform for embedded systems with self-x capabilities. That means, that ORCOS has to cope with dynamically changing behavior and has to be able to satisfy dynamically changing requirements on system service at runtime. Hence, besides the fine-grained offline configuration and customization, the operating software must implement methods for online reconfiguration itself [Stahl et al., 2014a]. In fact, online reconfiguration is realized in ORCOS twofold as it is not only restricted to react on dynamical changes, but, furthermore, is applied to implement self-optimization facilities to the operating system as well.

Reconfiguration can be triggered by different sources: external and internal requirement changes. Internal requirements are coming from the operating system itself based on self-optimization. Usually, the operating system's objective is to optimally manage the application's task and the resources e.g. in terms of resource consumption, memory management, scheduling strategies, etc. To meet these requirements, the operating system requires internal structures to monitor and analyze its own performance and to verify whether the optimization objectives are fulfilled. An appropriate architectural approach implemented in ORCOS will be discussed in section 8.2.

In this section, we concentrate on the foundations for online reconfiguration. Considering external requirement changes that originate from outside the operating system, which is either from the software or the hardware layer, it is easy to illustrate basic requirements on online reconfiguration: Due to a new software configuration, the implementation strategy of an OS service must be exchanged. To be able to satisfy such requirement changes, the operating system must provide alternatives concerning implementation. On the other hand, reconfiguration in software may lead to require an OS service that has not been provided by the operating system before. To satisfy this, the operating system needs structures to enable to load, exchange, extend, and activate OS components at run-time.

As ORCOS is intended to exhibit small footprints, it is not desired to load every possible system part on the target device. Online reconfigurability of the system requires the exchange or additional loading of tasks or system components at run-time. Therefore, ORCOS offers a Dynamical Task Loading framework which is described in section 8.1.2.

The basis for online reconfigurability of the operating system and the application tasks is provided by the Profile Framework which allows to switch between different configurations.

8.1.1 Profile Framework

Originally, the Profile Framework has been developed in the context of the Flexible Resource Manager (FRM) [78] to self-optimize resource consumption in resource restricted real-time systems by safe over-allocation of resources under hard real-time constrains. By introducing the Profile Framework, the FRM allows for alternative implementations of an application task in terms of resource requirements defined in *profiles*.

The Profile Framework follows the following principle: at each point of time exactly one profile of a task is active (see Fig. 8.1). A configuration c of the system is defined as configuration $c = (p_1, p_2, \ldots, p_n)$, with n being the number of running tasks τ and $p_1 \in P_1, p_2 \in P_2, \ldots, p_n \in P_n$ and P_i being the profile set of task τ_i. Each task must define at least one profile to be executed. For any task there may be available multiple profiles, i.e. versions with different parameters concerning nonfunctional properties. Selecting a specific profile is done due to dynamic decisions at run-time.

In the context of online reconfiguration of the entire system, we have enlarged the concept of a profile to be applied to any reconfigurable OS component. This encompasses all system entities:

- application tasks
- OS kernel components and services
- components of the self-x framework (e.g. reconfigurable Monitor and Analyzer as referred in section 8.2.)

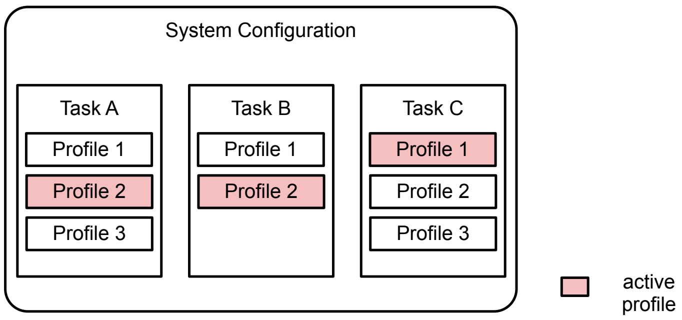


Figure 8.1: Profile Framework: Example for a system configuration

Profiles may differ concerning their resource demands, which resources are applied (e.g. a specific communication resource), the implemented algorithm (e.g. in terms of accuracy of the algorithm or the strategy), execution times, deadlines etc. A prerequisite for identifying a component as reconfigurable is the existence of alternatives, which in fact means the definition of at least two different profiles. Applying profiles to all OS components and the applications running on the system, all system parts become online (re)-configurable. Reconfiguration is realized by imply switching between the profiles. In order to maintain predictability and timeliness of the system, switching times and conditions are defined between the profiles.

For the reconfigurations, a controller module (see section 8.2) is responsible. It contains policies, restrictions and thresholds for decision making. Of course, a decision for reconfiguration must be checked against the system characteristics and the real-time requirements of the applications tasks. A reconfiguration must not harm the system service delivery and guarantee the compliance with the task's real-time deadline. If all the conditions are meet, the controller performs the reconfiguration.

8.1.2 Dynamic Task Loading

Usually, the ORCOS kernel's executable binary is loaded together with the tasks to be executed onto the target device. As a basic function towards online reconfigurability, ORCOS offers the ability to dynamically load additional tasks on the target at runtime.

As a matter of fact, real-time operating systems cannot accept adding new tasks unconditionally as they have to guarantee the schedulability of the tasks already present in the system. A feasibility check for the task set $J' = \{J \cup J_{new}\}$ consisting of the present task set J in conjunction with the new arriving task J_{new} is required in advance in order to ensure the resource and timing constraints of the system. ORCOS implements this feasibility analysis by the OROCS' `Task Loading Module`.

For ORCOS, tasks to be supplementary loaded reside in the form of their binaries in a task repository located on a remote server. Tasks in the repository are identified by a unique name. The connection to the task repository server is established by the ORCOS kernel internally if the kernel is appropriately configured. ORCOS offers a number of system calls: `getTasktable`, `preTaskloading`, `create_task_physicalMemory`, `create_physical_syscall` and `isDownloading` (for description of these system calls see Appendix A.7). Because system call are provided, the dynamical loading of a task at runtime can be initiated by other running tasks.

The scenario for dynamic task loading is illustrated by the sequence diagram in Fig. 8.2 in which the operating system modules are compromised into one ORCOS kernel object. The task

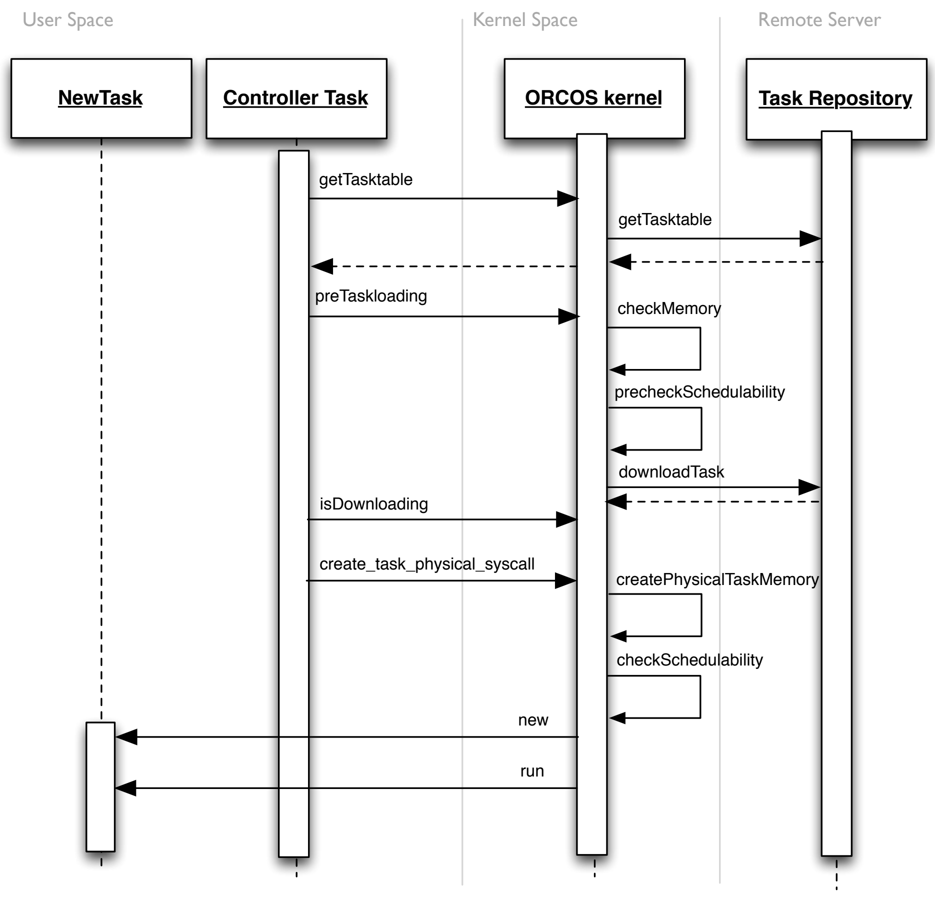


Figure 8.2: Dynamic loading of tasks at runtime

that implements the loading of other tasks (in Fig. 8.2 called `ControllerTask`) merely initiates the task loading procedure by calling the system call `getTasktable` with the according task name. By doing so, the`Task Loading Module` checks whether a task identified by the given task name exists in the repository, and it downloads and returns the Task Table of the task.

If the task exists, the operating system has to verify whether this task can be accepted to be executed. Even though the operating system modules are compromised into one ORCOS kernel object in Fig. 8.2, the individual modules are engaged in different subtasks of the acceptance test: In the first instance, the kernel's Memory Manager has to check whether the system has enough memory space available required by the task to be loaded on the system. The system call `preTaskloading` implements the searching for memory space for the new task based on the information delivered by the task's Task Table. If the task does not pass the memory check, the ORCOS kernel rejects the task loading and returns an error signal to the initiating task. It returns an OK signal otherwise and allocates some buffer for the task download. Then, the kernel performs an initial schedulability check to verify the task's execution feasibility. If the new task passes both, the memory check and the initial schedulability analysis, based on the information delivered by the Task Table, the kernel starts downloading the complete binary of the tasks.

The downloading of the task is proceeded in the background of the system. The initiating task can test the progress of the download by calling the system call `isDownloading` . After completing the downloading, an own memory space must be provided for the new task (by the calling `create_physical_syscall`). Then, the ORCOS kernel initializes the task.

Before being able to include the new task into the task set of ready tasks in order to be scheduled, a (more sophisticated) Schedulability Analysis is performed. The method for Schedulability Analysis is related to the applied Scheduler. If the task passes the Schedulability Analysis, it will be activated in order to be scheduled.

8.2 Basic Self-X Architecture

Usually, the workflow of a self-x system includes the following steps [49]:

1. Analysis of the current situation,

2. determination of objectives and

3. adaptation of the system behavior.

Hence, a real-time operating system with self-x capabilities must provide mechanisms and structures to enable the implementation of the self-x workflow. However, the adaptations to changes in the system must lead to robust system behavior in order to prevent that the system will run out of control.

The Organic Computing Initiative[3] has initiated the Observer-Controller Architecture[82] which was developed as an approach for self-organizing systems. The main objective of this architecture is to safeguard autonomous adaptations by maintaining the control of the system and its reactions on the dynamic adaptions. In order to realize self-x features with robust behavior in ORCOS, the ORCOS architecture was extended based upon the idea of the Observer-Controller Architecture.

The Observer-Controller Architecture builds up a feedback and controlling loop on the top of the a system under observation/control (SuOC). It introduces two main components: an Observer and a Controller.

The main task of the Observer, according to [82], is to identify the current state of a system and draw conclusions in terms of future states. As this task is not necessarily a trivial task, the Observer is split into subcomponents responsible for monitoring, pre-processing, data analysis, prediction and aggregation. The Observer is equipped with an Observation Model that specifies the selection of attributes to be monitored, the selection of appropriate analysis methods as well as the selection of the appropriate prediction methods. This Observation Model enables the Observer to be variable.

The main task of the Controller is implementing reconfiguration in terms of reaction of the system on the current system state and the potential future states. The decision making of the Controller is based on the output delivered by the Observer with respect to the context of the Controller's objectives. Changes initialized by the Controller may lead to changes in the requirements on the Observer. In order to react on that, the Controller is able to configure the Observer's behavior in accordance to its context purpose. The Observer offers therefore an interface to manipulate its Observation Model.

In order to adopt this architecture to ORCOS, we have identified that the Observer is taking over two distinct tasks that can be clearly separated from each other: monitoring and analyzing. Although strongly coupled within the Observer, we assume monitoring and analyzing to be self-contained tasks which can be executed in a timely decoupled manner [Stahl et al., 2014a].

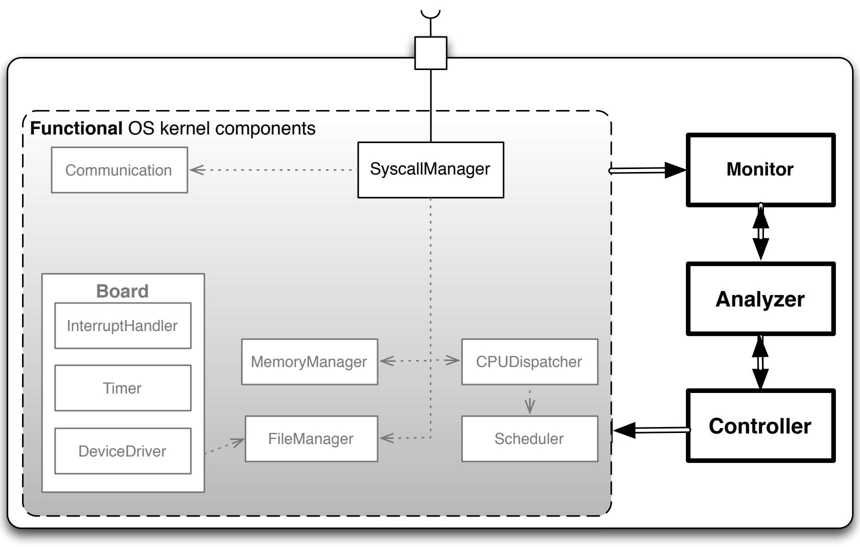


Figure 8.3: Generic Self-X ORCOS Architecture integrating a Monitor, an Analyzer and a Controller module.

Therefore, in the ORCOS Architecture the Observer was split up into two separate components, a Monitor and an Analyzer.

The resulting ORCOS architecture is presented in Fig. 8.3, containing Monitor, Analyzer and Controller [Stahl et al., 2014a]. These three components, namely Monitor, Analyzer and Controller, build up the so called ORCOS self-x framework that constructs a controlling loop of the system. The components of the self-x framework are integrated into ORCOS as configurable kernel components. In relation to the Observer-Controller Architecture, the ORCOS self-x framework modules are strictly separated from the functional OS kernel modules. By this separation, the self-x framework does not have any direct effect on the kernel's functional requirements so that the functional ORCOS modules can operate in an ordinary manner. Furthermore, it ensures that the functional OS modules can continue their work and will not break down if the self-x framework stops working. In order to guarantee real-time requirements of tasks, the ORCOS kernel modules proceed the task's request on the appropriate priority level while the self-x framework is operating in background using the system's idle times, in the first instance.

The self-x components initially define a generic framework. The introduced ORCOS modules provide only wrappers for implementation as well as interfaces between them. With these wrappers, the framework becomes open to different implementations developed for different objectives. The interconnection between the components is not only related to the interfaces for exchanging data (passing data from the Monitor to Analyzer, and delivering the output of the Analyzer to the Controller). Due to the ability of online reconfiguration we introduce an additional interrelation between the components: The strategy of the Analyzer is reconfigurable at runtime and can be exchanged by the Controller. The data collected by the Monitor is depending on the applied analyzing algorithm so that an exchange of the analyzing strategy in turn has an effect on the data aggregation of the Monitor.

8.2.1 Monitor

The Monitor is part of the self-x framework integrated into the ORCOS architecture. It allows different implementations as it initially provides only a wrapper for monitoring. Its main responsibility is identifying and selecting the appropriate information and collecting data. In accordance with the Observation Model of the Observer-Controller Architecture (discussed in the section 8.2), the monitored attributes can be customized based on the requirements defined by the Analyzer with respect to the global objective.

A specific implementation of the Monitor can be related to one OS kernel module as well as a combination of several kernel modules. Obviously, the implementation of the Monitor requires sophisticated internal knowledge of the system as well as the ORCOS modules in order to be able to extract the monitoring data. Furthermore, the Monitor can implement methods that prepare or even preprocess the collected data in accordance to the Analyzer's requirements. The implementation of the Monitor, in fact, relies on techniques established by scientific domains like data mining, time series generation, etc.

Real-time systems have limited memory and, hence, components of the operating system have strong restrictions on the amount of memory they can use. Besides the produced run-time overhead, these memory restrictions have to be considered when implementing the Monitor.

8.2.2 Analyzer

The Analyzer is located between the Monitor and the Controller within the self-x framework. It is responsible for analyzing the data that was aggregated by the Monitor with respect to the context purpose.

Like the Monitor, the Analyzer module in ORCOS is a wrapper for implementing different scenarios and analyzing algorithms. The output of the Monitor thereby deals as input of the Analyzer. So the Analyzer must be able to operate on the data - determined by the specification on the nature of data (see Chapter 2.2.1) - delivered by the Monitor. This leads to a strong interrelation and assigns a specific selection of data to be monitored to a certain analyzer algorithm. Usually, the algorithms implemented by the Analyzer belong to scientific areas like data mining, time series analysis and pattern matching, and/or apply mathematical or statistical methods, but also techniques from reinforcement learning, machine learning, artificial neural networks, and some others. The latter become more important if more differentiated conclusions such as predictions are required.

With a variety of different implementations, the Analyzer builds up a toolbox that may cover a broad range of objectives. Based on the global objective, the Controller selects the analyzing method out of the provided toolbox.

8.2.3 Controller

The central component for the run-time reconfiguration is build up by the Controller. It is the component that is responsible for decision making and initiating changes in the system. The responsibilities of the Controller are defined from different perspectives referring to three different controlling loops:

- control of the SuOC
- control of the self-x framework
- control of the objective function

On the one hand, the Controller is the closing component of the feedback loop for the SuOC. Its responsibility is to initiate a reconfiguration of the SuOC as a reaction to the evaluation procedure results. The results from the Analyzer are handed over to the Controller. The Controller has to examine the analysis results and evaluate which changes in the SuOC may be the best adaptation towards the desired direction determined by the objective function. These reconfiguration decisions on the SuOC are performed based on policies and pre-defined rules. Learning techniques in the decision making process can be applied in order to enhance the Controller's performance in finding better optimized configurations. Additionally, learning techniques introduce the ability of the Controller to adapt to dynamical changes, e.g. in the objective function. The reconfiguration itself is performed by the Controller on the basis of the Profile Framework (see section 8.1.1). Every application and every ORCOS module specifies profiles with different characteristics that are known to the Controller. Profiles match policy specifications of the Controller. In order to perform a reconfiguration, the Controller's task is to identify the appropriate profiles that cover the policies reflecting the objective function. Then, a reconfiguration by the Controller is realized by just switching between profiles of dedicated system modules and/or application tasks respectively. As the Profile Framework supports reconfiguration under real-time constraints [78], a reconfiguration of the SuOC by the Controller will only be initiated if all timing restrictions can be guaranteed.

However, the Controller is not only responsible for the configuration of the SuOC. It is part of the self-x framework that is reconfigurable itself. In fact, the Monitor and the Analyzer (see sections 8.2.1 and 8.2.2 above) offer interfaces to the Controller for configuration. The Controller uses these interfaces to select an appropriate analyzing method out of the provided toolbox with regard to the objective function. The choice of the analyzing method in turn has effect on the attributes selected for observation by the Monitor.

The objective function is the main reference to measure the quality of the system performance. Of course, the system may self-adapt its behavior because of dynamical changes. Changes in system behavior as well as environmental changes in turn may lead to variations or even to novel requirements on the objective function. The Controller may implement self-learning methods in order to adjust the objective function by itself. However, as a measure of quality, the Controller offers interfaces for the objective function to be determined by the user or developer as well.

Summing up, the Controller guides the self-x process of the entire system from three different viewpoints, having the impact on the system to be customized to each individual scenario.

8.3 Architecture for Anomaly Detection

The basic self-x framework was integrated into ORCOS to form a basis for self-x properties introduced in ORCOS. One application of the self-x framework is illustrated in [25] (as part of a Master Thesis) where it was shown how the framework has been applied to implementing self-healing components by using the example of device drivers.

For the purpose of implementing the Online Anomaly Detection approach, we use the self-x framework as a basis. In fact, the implementation of Online Anomaly Detection is another specific application of the generic framework and is illustrated in Fig. 8.4. As described in section 2.1, Anomaly Detection refers to the ability of detecting deviations by analyzing data obtained by observations. Additionally, referring to chapter 7, the main goal of this thesis is to present an approach for identifying anomalies in self-x systems and classify them by using ideas from Danger Theory.

The concept of our approach is based on Dendritic Cells which are the responsible entities for monitoring the behavior of the applications and therefore have to be be integrated into the Monitor. The behavior of the applications is constructed out of the system calls invoked by the applications which leads to monitoring of system call data within a System Call Monitor. The principle part of the Online Anomaly Detection mechanism is integrated into the Analyzer, consisting of a Behavior Knowledge Base based on this extracted data, the detection of anomalies integrated into the Behavior Knowledge Base construction method, and a classification module considering input signals delivered by the System Health Monitor.

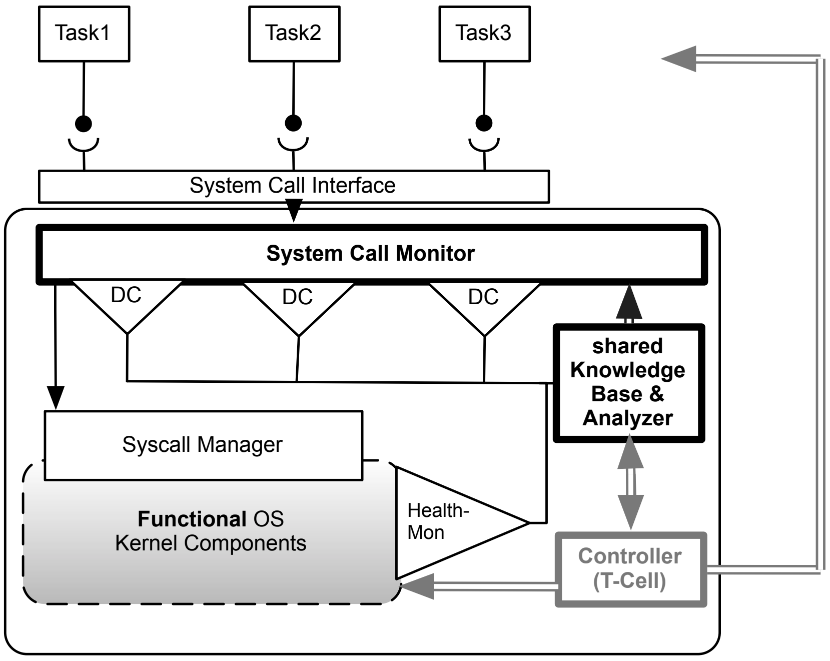


Figure 8.4: ORCOS Architecture integrating Online Anomaly Detection modules.

The objective of Online Anomaly Detection is to detect anomalies and classify them. In case of a potential detected danger, Artificial Immune Systems provide the principle of the so called immune response as a reaction on this danger. From AIS viewpoint, the immune response is initiated by T-Cells, which in the ORCOS self-x framework may be represented by the Controller. It is important to emphasize here, that this thesis is not concerned with reactions in case of detected anomalies classified as suspicious or even malicious. Therefore, we are mainly interested in the parts of the Monitor and Analyzer for the implementation of the approach.

Appropriate reactions on anomalies identified in task behavior are difficult to be specified generically. Possible reactions may be a shutdown of the task with potential dangerous behavior, its restart or a resume to the last checkpoint of stable applications state, but also reconfiguration by switching into another profile or applying application-specific self-healing mechanisms. However, there is no general answer how to react on unstable states as e.g. a shutdown of a critical application lead to severe consequences or even failures of other system parts but might be the best solution in another case. Furthermore, the application does not always need to be the source of the problem. Unstable application behavior may also be caused based on failures in the underlying system, namely faulty operating system modules or erroneous hardware devices. Hence, more application-specific knowledge is required as well as considering the entire system context to judge what is the best reaction on an unstable application behavior detected. Even if forming an interesting research topic, this problem is far beyond the scope of this thesis. Consequently, for the general purpose of implementing the presented approach for Online Anomaly Detection, an implementation of the Controller is not in the scope of this approach so that investigations will be omitted (Controller part of the architecture is therefore colored in grey in Fig. 8.4).

8.4 System Call Monitor

The behavior of tasks is defined by the system calls or more precisely the sequences of system calls that are invoked by the tasks as system calls providing the only interface between the user space tasks and the kernel. The System Call Monitor is responsible for extracting the appropriate information that are then aggregated to construct a Behavior Knowledge Base for anomaly detection. The anomaly detection is dedicated to work in online manner. Therefore, the system call informations must be recorded at the time when a system call happens to realize online monitoring. In order to do so, the System Call Monitor is located directly in between the System Call Interface and the System Call Manager (see Fig. 8.4). Fig. 8.5 shows in a sequence diagram how the System Call Monitor is integrated into the execution of a system call (see Fig. 6.4 in section 6.3.3 for comparison).

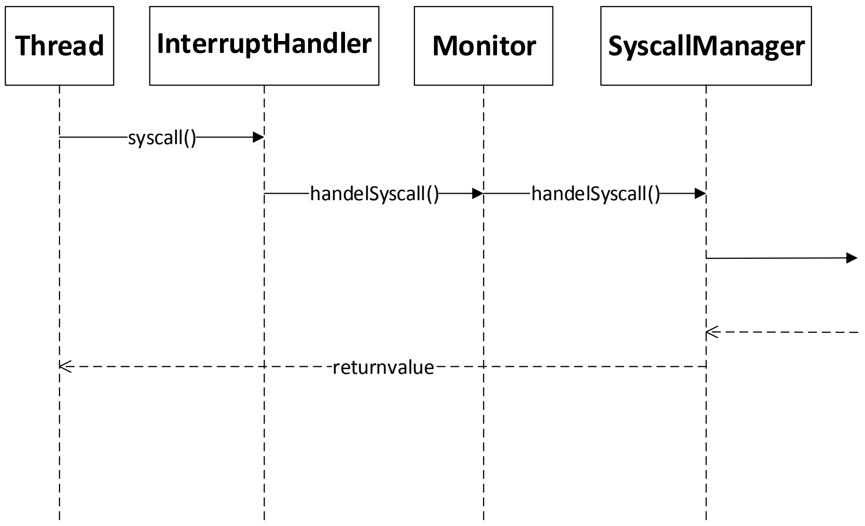


Figure 8.5: Integration of monitoring into the OROCS System Call Manager

The concept of integrating self-x properties into ORCOS requires that all modules of the operating system are online reconfigurable. This implies the self-x framework itself, and hence, the Monitor component. In order to comply with this concept, the System Call Monitor is designed to offer a framework for online reconfigurable monitoring. The idea of this framework is to supply a broad range of analysis algorithms by the same monitor which is generic and independent of any specific anomaly detection implementation. Flexibility is required, as different algorithms use different parameters for evaluation. Some of the anomaly detection algorithm considering system calls use just the system calls themselves, while others use system call arguments, return addresses of the stack or the program counter. The different approaches have been summarized in Chapter 4.

Even though the Online Anomaly Detection specified in Chapter 7 is the core approach to be evaluated in this thesis, it should be possible to integrate other approaches without any great programming or configuration effort in order to exploit their qualities and to provide a base for making the different approaches comparable. That means that whenever the applied analyzing algorithm will be exchanged, the monitor shall be set up to the according requirements. The detailed concept and implementation of the Monitoring Framework has been developed in the context of a Master Thesis that can be found in [92]. The latter of this section illustrates the basic properties of the System Call Monitoring Framework.

The following information can be extracted (up to now) from a system call:

- **Thread Identifier**: Tasks in ORCOS are composed of threads that build up the basic units of execution. Hence, it is the thread that actually calls a system call. This, in turn leads to the fact that system call information is collected at thread level. A system call is assigned to a particular thread by its identifier.
- **System Call ID**: Each system call provided by the system has its unique own identifier. The assignment of system calls to their according IDs is listed in Appendix A.
- **System Call Arguments**: Some of the system calls specify parameters to be passed to the kernel method. Obviously, the list of the arguments belonging to the system call can be recorded.
- **Stack Return Address**: An invocation of a system call leads to a context switch as the system call is assigned to a kernel function executed in kernel mode. After execution, the system call returns to the calling stack address. This stack return address is recorded.
- **Acquired Resources**: ORCOS treats files, stream devices, communication devices, sockets, etc. as resources assigned to a task with each one of them having a unique address in the filesystem. The access to resources is encapsulated within system calls and recorded with the associated system call.
- **Time Stamp**: The system call is invoked at a particular time; this time stamp is recorded.

Real-time systems have limited memory and, hence, the operating system has severe restrictions concerning the amount of memory it can use. These restrictions apply to the monitor as well. As not every analyzing algorithm processes all this information, unnecessary overhead in terms of run-time and memory space would be produced if the monitor would store all the data connected to a system call. Therefore, the monitor is designed to be reconfigurable at run-time in order to modify which parameters are monitored in accordance to the analyzing algorithm. The System Call Monitor defines *Monitoring Modes* to be configured in order to control the memory usage of the monitor. The recorded data is stored in a *Monitoring Database* in order to be aggregated and processed by the analyzing method. The Monitoring Database is configurable at run-time as well. To be able to control the Monitoring Modes and the Monitoring Database, the System Call Monitor offers a Monitor API. These three components of the Monitoring Framework will be described in short in the following section.

As the monitor intercepts the system call execution in order to record the data, it introduces additional run-time overhead into the system call handling. The run-time overhead as well as the memory usage caused by the monitoring is evaluated and discussed in section 9.1.

8.4.1 Monitoring Modes

The System Call Monitor is able to extract the system call information listed above. Storing all this data for every system call would lead to the waste of memory if the data is not required by the anomaly detection algorithm. To analyze system call sequences, any kind of algorithm requires at least the following parameters: *Thread Identified*, in order to assign the system call to the calling thread, *System Call ID* in order to record the system call itself and *Time Stamp* in order to ensure the correct ordering of the monitored system calls.

The remaining parameters currently supported are to be selected according to the requirements of the applied analyzer. Therefore, the monitor defines modes by which the selection of parameters to be recorded can be modified at run-time in accordance to requirement changes of the analyzer.

The parameters that are currently supported by the modes are:

- system call id
- system call arguments
- return address stack
- task resources

We use a bitmap to represent the mode, with each bit representing a specific parameter that can be monitored. This allows us to monitor multiple parameters at the same time, which may be useful if an analyzing algorithm needs e.g. both *system call id* and *system call arguments* as input parameters. If the analyzer will be reconfigured in such a way that it relies on a different input data set, the reconfiguration of the monitor is invoked by changing the *Monitoring Mode*. (The mode can be set to *zero* which deactivates the *system call id* and consequently, deactivates the entire monitoring module. This option is defined for some critical cases that require the self-x framework to be switched off in order to preserve resources for the functional part of the OS.)

8.4.2 Monitoring Database

The extracted system call information belonging to a system call and determined by the mode of the System Call Monitor are encapsulated within a System Call Record (SCR). The collected information (System Call Records) are passed from the Monitor to the Analyzer in order to be examined. The transfer of the data can be realized in different procedures: On the one hand, the Analyzer is supplied with input data as soon as the data is recorded. On the other hand, the Analyzer requests the data at the point of time when it requires it. Complying with our idea of timely independent components, the latter version is the one preferred by this approach. However, this means, that as the System Call Records will not be processed by the Analyzer immediately, the Monitor is dedicated to preserve this data until it is requested. Due do timely decoupled execution, the Monitor is intended to store multiple SCRs as the Monitor does not have any information about the execution schedule of the Analyzer. Hence, it implements a Monitor Database as a data structure to hold and manage the aggregated system call information.

In our idea of applying Dendritic Cells from Danger Theory, Dendritic Cells are responsible for the observation of one task. In fact, it is the thread that forms the executable unit in ORCOS so that per task at least one thread has to implemented. (In practice, almost all of the example

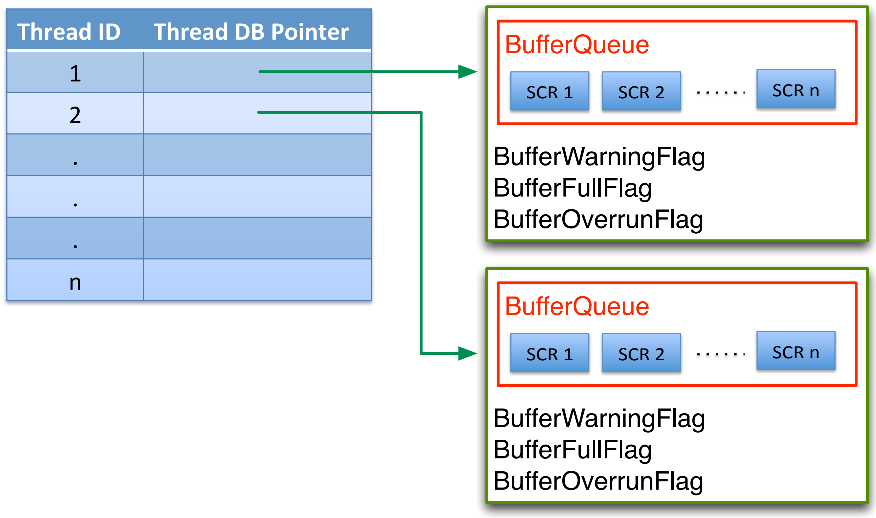


Figure 8.6: Architecture of the System Call Monitor Database

applications existing in ORCOS contain exactly one thread.) Thus, a DC is intended to collect system call information triggered by a particular thread. In order to support this requirement, the Monitor Database is organized on thread basis. For each thread, the SCRs are stored in an individual buffer queue. The architecture of the Monitor Database is shown in Figure 8.6.

The memory amount is governed by the selection of the monitored parameters which is reflected by the size of the System Call Record, but more essentially it is governed by the frequency of the system call invocations in correlation with the frequency of requesting the system call data from the Analyzer. Therefore, estimating the memory usage of thread buffers is a complex issue. The Monitoring Framework thereby offers two different options for buffer allocation: static and dynamic. This is to be configured by SCL offline, so that the decision which buffer allocation method to choose must happen at the system's compile time. Both options have their pros and cons:

- **static buffer configuration**: If the buffer is configured to be a statically allocated buffer, the size of the buffer has to be defined offline. The buffer is implemented as a local circular queue with a number of System Call Records that can be preserved. Static allocation guarantees a constant time for data storing. If the queue is full and the System Call Monitor is up to capture another system call, then the oldest entry in the buffer will be overwritten. Therefore, it is essential to estimate a safe buffer size at compile time. For this purpose, the frequency of system calls per thread, and the schedule of the analyzer must be know in advance.

- **dynamic buffer configuration**: If the monitor uses a dynamic buffer configuration, the buffer size can be configured individually on a per thread basis. The monitoring framework provides an early warning mechanism that raises a flag whenever the buffer is approaching full utilization. Then, the maximum buffer capacity can be changed at runtime. Thus, the dynamic buffer configuration enables precise buffer control, with efficient utilization of the available system memory.

  However, while using the dynamic strategy, the monitor always needs to allocate memory for the system call record first before it can store the data gathered. This allocation requires an additional time overhead which might not be constant.

The behavior of the monitor differs for each approach as the internal representation, data storage and retrieval strategies differ. While the static strategy needs constant time for storing a System Call Record, in the dynamic strategy it is depending on the performance of the Memory Manager applied. In contrary, when data is requested by the Analyzer, using a static buffer requires coping the data which results in run-time and space overhead while for the dynamic data, it is merely a handing over of the pointer of the according data set. The decision which buffer allocation method to choose is in fact a trade-off between the efficient storage and constant allocation time, taking into account memory wastage and differences in run-time demands considering the retrieval of system call data.

As real-time systems require to be fully deterministic, the dynamic strategy does not appear to be applicable in this context. However, recent research in this field has resulted in approaches like the Two Level Segregated Fit Allocator [75] that is capable of dynamically allocating memory in constant-time. These recent developments may suggest the use of such dynamic memory data structures in modern real-time systems in the near future.

8.4.3 Monitor API

The Monitor is designed to be reconfigurable at runtime in terms of its mode identifying the parameters to be monitored as well as, when using the dynamic buffer configuration, in terms of setting the buffer size. Furthermore, it is dedicated to provide input data to the Analyzer. In order to offer interfaces to these functions, the monitor defines an API with the following operations:

- change the monitor mode
- set the maximum buffer size for a thread
- set the buffer warning threshold for a thread
- check if a threads buffer is full
- check if a threads buffer warning flag is raised
- delete all the data entries for a thread
- get the first entry from a thread buffer

The specification of the methods offered by the Monitor API can be found in Appendix B.

The System Call Monitor has been integrated into the ORCOS Architecture in oder to support anomaly detection approaches that operate on system call data. The System Call Monitor was designed in a generous manner offering configurable modes and parameters so that it can be easily customized to the various requirements of different anomaly detection methods. During the entire design and development of the monitor, the major requirement on the monitor was followed to be as less intrusive as possible and have a minimum impact on the schedulability of the tasks in the system. The System Call Monitor has been developed and evaluated as a stand-alone module in ORCOS within a Master thesis by Gavin Vaz [92]. This Master thesis (supervised by me) was implemented in a strong interaction with this thesis as its intention was to provide input data for the anomaly detection approach presented in this thesis. An exhaustive evaluation of the System Call Monitor has been performed and described in [92]. A summary of the stand-alone evaluation of the System Call Monitor is provided in Section 9.1.

8.5 Behavior Knowledge Base

In Chapter 7, the so called Dendritic Cells coming from Danger Theory have been introduced to observe and collect behavior information about the tasks. Thereby, each task is assigned a DC. The behavior to be observed has been defined as sequences of system calls executed by the task (see Definition of Behavior in Chapter 7.1.1). Hence, these are the information about a task collected by a DC. The lifetime of a DC has been assigned to a period of a task's execution in which the DC collects the information in order to be evaluated. Referring to the DC's lifetime, a behavioral sequence is determined by the execution of an instance of a task and is therefore related to a task's period.

According to the applied model of Danger Theory, all the occurring behavioral sequences collected by DCs have to be evaluated and are then transferred to the *thymus* in order to be conserved. The thymus is then responsible for memorizing the entire behavioral history which deals also as an indicator input for anomaly detection. For the immunological entity of the thymus, in fact, the computational counterpart is a storage of history that we call Behavior Knowledge Base. The objective of the Behavior Knowledge Base is to store all information collected by DCs. The Behavior Knowledge Base is intended to contain all the behavioral sequences in order to preserve the entire execution history of the tasks, and thereby to build the basis for establishing a reliable knowledge about a tasks' (normal) behavior.

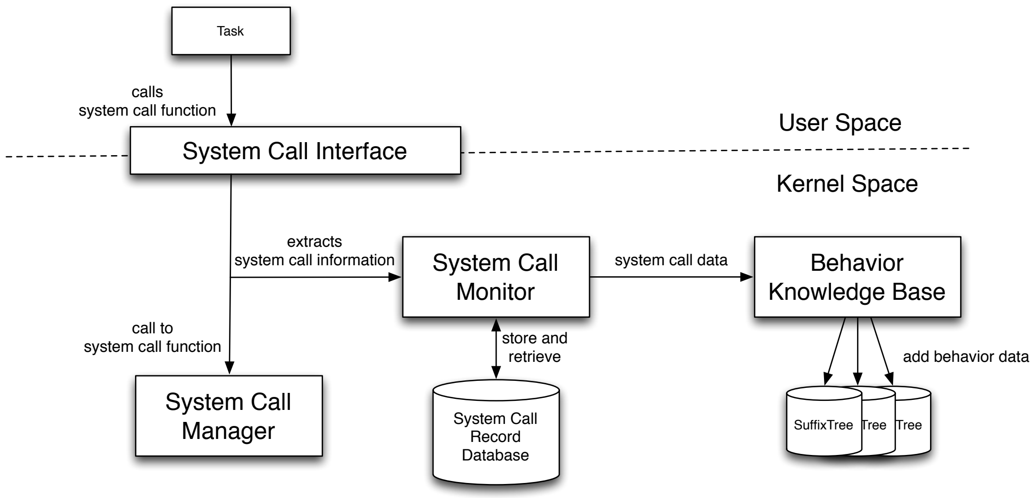


Figure 8.7: Data flow of system call data through System Call Monitor to Behavior Knowledge Base

In ORCOS, the system call sequences are extracted by the System Call Monitor (described in the section above, Section 8.4) and stored as System Call Records in the System Call Monitor's database. The System Call Monitor provides the source of input for behavior collection of DCs that is then passed to the Behavior Knowledge Base. The data flow of the system call data is shown in Figure 8.7.

The extracted input data coming from the System Call Monitor are preprocessed and transformed into the particular representation structure in the Behavior Knowledge Base. As described in Chapter 7.3, Suffix Trees have beed chosen as the appropriate data structure to represent the Behavior Knowledge Base. With its characteristics described in Section 5.2, Suffix Trees offer great potentials to construct a knowledge base for behavior patterns consisting of sequences of system calls, as they enable to process the sequences in a symbol-by-symbol manner (using Ukkonen's algorithm, see Section 5.2.3).

This section introduces the Suffix Tree implementation with the adaptations made on the original data structure that have been necessary to make Suffix Trees applicable to build up a Behavior Knowledge Base for the tasks executing on the RTOS.

8.5.1 Internal Behavior Sequence Representation

The System Call Monitor extracts the invoked system calls with the related information that have been configured in the monitor's mode. For the purpose of our approach, it is only required to extract the system call IDs which are stored in the System Call Monitor Database. (The IDs are unique identifiers of the system calls. The assignment of system calls to their according IDs is listed in Appendix A.) The monitor's database is organized in such a manner that all System Call Records (information related to one system call) associated with one task are stored in one buffer queue (see Section 8.4.2). Therefore, the system call sequence belonging to one task's instance can be easily extracted from the System Call Monitor Database. The system call sequences is a sequence of system call IDs.

In the database, the buffer queue is continuously filled with entries of the system calls triggered by a task. The System Call Monitor that is connected to the System Call Interface has no knowledge about processes taking place in the operating system, such as context switches, scheduling decisions etc. Context switches, however, could be identified by the System Call Monitor based on the fact that the task ID of the monitored system calls changes. Even though, this information does not enable the System Call Monitor to distinguish between a preemption or a task instance's termination. However, for the purpose of the anomaly detection, it is important to identify the end of a task instance's execution. The major reason for that is based on our idea of DC that related to a task's period. But it is also necessary for the applied data structure (the Suffix Trees) to mark the end of a sequence in order to enable differentiation between two sequences in which one is the prefix of the other (further subsections are illustrating this problem). In general, Suffix Trees use unique symbols depicted by the $-symbol to clearly identify the ending of a sequences. In ORCOS, however, the last system call of a task instance is the call of the `terminate()`-method, or more precisely the `thread_exit`-system call, even if not explicitly implemented by the task developer. Therefore, every sequence of system calls triggered by one task instance is extended by the unique symbol having the ID 16 for determining the sequences' ending. Then a behavioral sequence S, collected by a DC, for task i and its jth execution has the following structure (s_x being the system call IDs of the invoked system calls):

$$S_{i,j} = (s_1, s_2, \ldots, s_n, 16)$$

The Sequence Ending Symbol is generated internally and included into the buffer queue at the point of time a thread's `terminate()`-method is called (the $ is synonymously used for the system call ID 16 in the remaining part of this chapter).

8.5.2 Internal Implementation of the Suffix Tree

Referring to Chapter 5.2, a Suffix Tree consists of a root node that is its entry point. From the root node, paths descend that represent the suffixes of a particular string or sequence. The paths consist of labeled edges connecting nodes as branch points of the path and leaf nodes as ending points of a path. The labels on the edges of one path chained up form the symbols of the sequence's suffix. For each suffix one path exists leading into a leaf node.

Our Behavior Knowledge Base is intended to contain not only one behavioral sequence but moreover all behavioral sequences occurred. Therefore, for the implementation of the Behavior Knowledge Base, we comply with the concept of the Generalized Suffix Trees (see Section 5.2.4) allowing multiple sequences to be inserted into one data structure. For our purpose, the concept of Suffix Trees has a major drawback by reason of concatenated labels on the edges:

1. Suffix Trees are built up for the purpose of exact matching of sequences. For anomaly detection, exact matching is required as we are particularly interested in identifying deviations from the previously observed behavior. We expect the behavior of task instances to be quite similar but not always identical to the behaviors of former executions of the task. Similar behavior leads to similar sequences not being identical and hence, generates novel sequences. Inserting novel sequences into a Suffix Tree requires an extension of the Suffix Tree that, depending of the variety of a task execution flow, may become common. Referring to Section 5.2.2, an extension of a Suffix Tree is realized by splitting up the label after the last symbol that is matching the sequence to be inserted and inserting a novel branch node at that place. A new leaf node is generated and connected with the new branch node by an edge. The label of the new edge is initialized with that part of the sequence that did not match the already existing labels (starting at the first symbol mismatching).

   From the programming point of view, such extension of the Suffix Tree is difficult and expensive. It requires the following steps:

   Step 1 identifying the position of the last matching point in the label

   Step 2 generating two new labels by splitting the original label of the affected edge: one for the first part that is matching the new sequence and the second label for the part of the affected label that does not match the new sequence.

   Step 3 generating a new node

   Step 4 connecting the affected edge (which label has been split) to the newly generated node and setting the first part of the label (the matching part) as the label of this edge

   Step 5 generating a new edge that connects the new node and the former destination node of the edge and labeling that edge with the second part of the label of the split edge (the mismatching part).

   Step 6 generating a new leaf node for the new (mismatching) part of the sequence

   Step 7 generating a new edge labeled by the mismatching part of the sequences and setting this edge as a connector of the new branch node and the new leaf node.

   Obviously, this has to be done for all the suffixes of the sequence as in a Suffix Tree any extension in a path requires to be performed for all the paths containing a suffix of that path. The run-time complexity is mostly related to the splitting of the labels and is depending on the internal implementation of the labels: The labels may be realized as arrays which might be the most efficient data type from memory viewpoint as well as concerning real-time requirements (static size, deterministic access to entries). If labels are implemented by arrays, then while splitting 2 new arrays have to be generated. The initialization of these two arrays is then realized by copying from the original array to the two new ones. This has to be processed for all the paths containing suffixes of the matching part of the sequence which leads to much run-time overhead. Using arrays, however, has another drawback that is related to its static size. It is difficult to estimate in advance the size of a label at one edge. Furthermore, it becomes more complicated if an array was chosen too small and has to be enlarged when it has to preserve more entries that previously expected.

An approach to overcome this problem is to use dynamic data structures such as a linked list to realize the concatenation of labels. Linked list solve the problem of the complexity of splitting up the labels. In linked lists, a splitting would only require just a redirection of pointers instead of copying the existing entries. However, there is another reason why concatenation of labels at edges for our purpose becomes unhandily:

2. In the approach proposed in this thesis, the execution of a system call sequence but also each execution of a system call is intended to be analyzed individually in terms of its effects on the (operating) system state. Hence, further information (such as frequency of occurrence and classification of effect) not only have to be assigned to a (sub-)sequence but also to each system call. Using a linked list for edge labels may allow to integrate these information to each symbol in the label by appropriate design of the symbol object. However, preserving the same type of information for (sub-)sequences as well as individual system call leads to high redundancy of data which is not conform with the requirements of real-time operating system having restricted resources.

To overcome these problems, we have decided to internally implement the Suffix Tree, in fact, in form of a *Suffix Trie* (see Section 5.2.5). The main difference between Suffix Tree and Suffix Trie is that in Suffix Tries an edge is labeled by only one symbol as depicted in Figure 8.8.

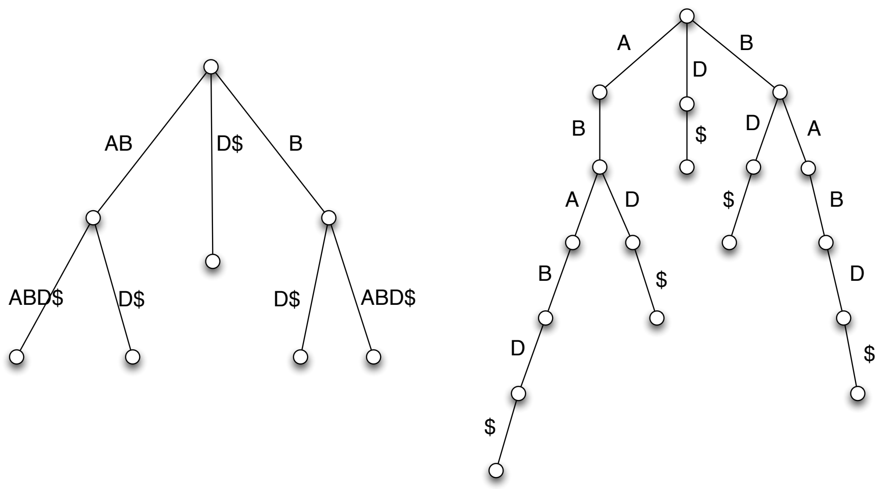


Figure 8.8: Comparison between a Suffix Tree (left side) and a Suffix Trie (right side) for the sequence $ABABD\$$.

Of course, applying Suffix Tries enlarges the number of nodes in the data structure. But on the other hand, a path extension is easier to be programmed for Suffix Tries as it only requires to generate a new node and an edge connecting to an existing node. This simplification is essential for run-time purposes. Additionally, it enables to integrate the classification information into each node and thereby allows to equip each individual system call as well as an entire (sub-)sequence with classification information. Besides this difference, Suffix Tries have the same characteristics and features as the Suffix Trees.

Now the data structure consists of the following elements:

- one root node
- passthrough nodes (with only one descendant)
- branch nodes (with more then one descendants)
- leaf nodes
- edges connecting the nodes with each other.

On grounds of programming simplification, the labels of the edges are implemented as an attribute of the nodes. For complete sequences (in terms of representing a behavior sequence of a task instance complete execution), leaf nodes will have to be labeled by the Sequence Ending Symbol \$. Only at the point of time when sequences are not completely inserted, leaf nodes are able to contain other labels then the Sequence Ending Symbol. The Suffix Tree implementation consists of three classes (as depicted in the UML class diagram in Figure 8.9):

`TreeNode` represents the implementation of the nodes of the tree. Each node is assigned a `value`-attribute for the system call ID, the `nodetype` determining the type of the node in the Suffix Tree, a reference to the node's `father`, an array holding the `sons` of the node and some further helper attributes that are required for construction and searching purposes. The specified methods are offered for the access to the values of the listed attributes. The edge implementation is realized by the pointers to `sons`-nodes, as there are no information required to be assigned to an edge due to fact that the edge label integrated into the node itself in forms of the attribute `value`.

`SuffixPath` is a helper class designed to store paths of nodes. A `SuffixPath` is dedicated to be used to store such nodes that have been reached during construction of and/or searching in the Suffix Tree. Its main ingredient is an array of references to `TreeNodes` that can be accessed by the offered methods.

`SuffixTree` is the main class of the Suffix Tree implementation. It holds references to the `root`-node, the array of `leafnodes`, `branch point` and the according counter variables `numleafnodes` and `numbranchpoint`, a reference to the sequence `initSeqPointer` and its `length`. The class `SuffixTree` offers one constructor to immediately construct the Suffix Tree for a previously completely available sequence by `SuffixTree(int* initSeqPointer, int length()` and one constructor to initialize an empty Suffix Tree (`SuffixTree()`) in such cases when the sequences is not (completely) available. Furthermore, methods are defined to construct the Suffix Tree by `addNode(...)`, `addNodeToSuffixTree(...)`, `insertSeq(...)`. The array `pathpool` is defined to store references to objects of `SuffixPath` and is designed as a helper attribute to remember the paths of nodes reached during the construction and/or searching. The remaining attributes and methods are further helper attributes and methods defined for the construction and searching purpose and will not be introduced in details here.

The detailed construction method is explained in Section 8.5.4.

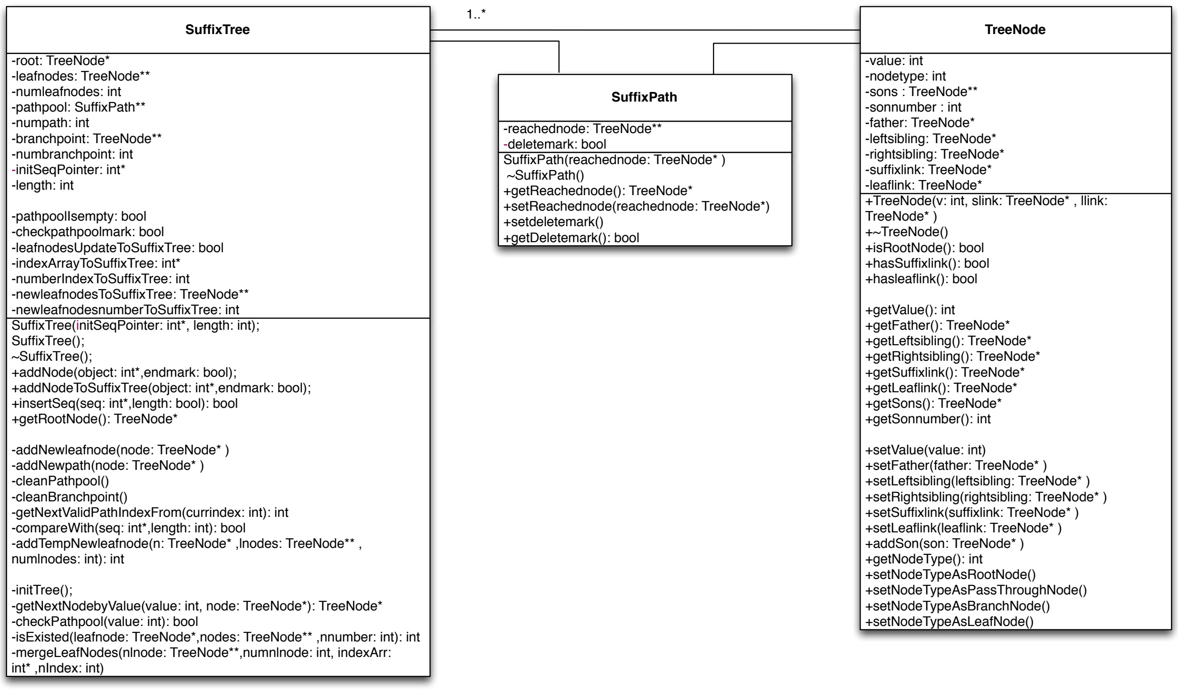

1..*
SuffixTree
-root: TreeNode*
-leafnodes: TreeNode**
-numleafnodes: int
-pathpool: SuffixPath**
-numpath: int
-branchpoint: TreeNode**
-numbranchpoint: int
-initSeqPointer: int*
-length: int
-pathpoolIsempty: bool
-checkpathpoolmark: bool
-leafnodesUpdateToSuffixTree: bool
-indexArrayToSuffixTree: int*
-numberIndexToSuffixTree: int
-newleafnodesToSuffixTree: TreeNode**
-newleafnodesnumberToSuffixTree: int
SuffixTree(initSeqPointer: int*, length: int);
SuffixTree();
~SuffixTree();
+addNode(object: int*,endmark: bool);
+addNodeToSuffixTree(object: int*,endmark: bool);
+insertSeq(seq: int*,length: bool): bool
+getRootNode(): TreeNode*
-addNewleafnode(node: TreeNode*)
-addNewpath(node: TreeNode*)
-cleanPathpool()
-cleanBranchpoint()
-getNextValidPathIndexFrom(currindex: int): int
-compareWith(seq: int*,length: int): bool
-addTempNewleafnode(n: TreeNode* ,lnodes: TreeNode** , numlnodes: int): int
-initTree();
-getNextNodebyValue(value: int, node: TreeNode*): TreeNode*
-checkPathpool(value: int): bool
-isExisted(leafnode: TreeNode*,nodes: TreeNode** ,nnumber: int): int
-mergeLeafNodes(nlnode: TreeNode**,numnlnode: int, indexArr: int* ,nIndex: int)
SuffixPath
-reachednode: TreeNode**
-deletemark: bool
SuffixPath(reachednode: TreeNode*)
~SuffixPath()
+getReachednode(): TreeNode*
+setReachednode(reachednode: TreeNode*)
+setdeletemark()
+getDeletemark(): bool
TreeNode
-value: int
-nodetype: int
-sons : TreeNode**
-sonnumber : int
-father: TreeNode*
-leftsibling: TreeNode*
-rightsibling: TreeNode*
-suffixlink: TreeNode*
-leaflink: TreeNode*
+TreeNode(v: int, slink: TreeNode* , llink: TreeNode*)
+~TreeNode()
+isRootNode(): bool
+hasSuffixlink(): bool
+hasleaflink(): bool
+getValue(): int
+getFather(): TreeNode*
+getLeftsibling(): TreeNode*
+getRightsibling(): TreeNode*
+getSuffixlink(): TreeNode*
+getLeaflink(): TreeNode*
+getSons(): TreeNode*
+getSonnumber(): int
+setValue(value: int)
+setFather(father: TreeNode*)
+setLeftsibling(leftsibling: TreeNode*)
+setRightsibling(rightsibling: TreeNode*)
+setSuffixlink(suffixlink: TreeNode*)
+setLeaflink(leaflink: TreeNode*)
+addSon(son: TreeNode*)
+getNodeType(): int
+setNodeTypeAsRootNode()
+setNodeTypeAsPassThroughNode()
+setNodeTypeAsBranchNode()
+setNodeTypeAsLeafNode()

Figure 8.9: Class Diagramm for Suffix Tree implementation.

The implementation of the Suffix Tree in form of a trie is only realized internally (due to restrictions originating from requirements on runtime and memory overhead originating from real-time systems), and it does not have any effect on the design of the anomaly detection approach presented in this thesis. Therefore, we will continue referring to Suffix Trees as the data structure for our Behavior Knowledge Base in the following parts of this thesis. We will only emphasize the real internal realization by Suffix Tries if it becomes important.

8.5.3 Behavior Knowledge Base Initialization

The behavior for each running task in the system is assumed to be unique although the individual executions of task instances may vary. Mainly, it is determined by the control flow executed by the application task that leads to a particular system call sequence representing an instance's execution. With the Behavior Knowledge Base, we aim to establish a stable and reliable knowledge of the (normal) behavior for each task in order to enable an individual evaluation of a task's behavior.

Therefore, each task requires its own Behavior Knowledge Base that is represented by a Suffix Tree. Hence, for each task a Suffix Tree is generated at the task's initialization time (by using the constructor for sequences not yet available `SuffixTree()`). At initialization, the Suffix Tree is empty consisting of only (a pointer to) the root node. The Suffix Tree remains empty as long as the task's first instance is not executed.

The collection of the single task's Behavior Knowledge Bases constructs the shared Behavior Knowledge Base of the anomaly detection framework.

8.5.4 Construction of the Behavior Knowledge Base

Before a task is executed for the first time, its Behavior Knowledge Base has been generated at the task's initialization by creating an empty Suffix Tree assigned to this task. In fact, the first execution of a task (which is its first instance) does not differ from any other execution of the task's instance, even if an initial execution may proceed some initialization functions. From the operating system's point of view, each task instance is treated equally with respect to its characteristics such as WCET, period, priority assignment, assigned resources etc. However, concerning the Behavior Knowledge Base, the first execution of a task differs in such a manner that the Behavior Knowledge Base is empty at the point of time the task starts its execution. The Suffix Tree has to be constructed "from scratch" for the task based on the behavior of the first (actual) task instance.

Each system call executed by the task is monitored by the System Call Monitor and passed to the Behavior Knowledge Base as depicted in Figure 8.7. As the system calls occur in a sequential manner, they have to be processes one after the other and included into the Behavior Knowledge Base that is represented by the Suffix Tree. According to the consecutive emergence of the system calls, the construction of the Suffix Tree for the behavioral sequence of a task is implemented in a symbol-by-symbol manner on the basis of the algorithm presented by Ukkonen (see Section 5.2.3). For each occurring system call, its ID is extracted (associated with a task/thread ID and passed over to the Behavior Knowledge Base. The Behavior Knowledge Base holds all the Suffix Trees (one for each task). The system call IDs are inserted into the associated Suffix Tree of a task by calling its method `addNode(int* value, bool endmark)`. Then, the Suffix Tree is extended by this new sequence symbol right after the last symbol of the sequence that has been inserted before.

To show the procedure of Suffix Tree construction, we take a look at a simple example for an application source code, but by ignoring the semantics of that application. As we are only interested in the system calls executed by the application task, for simplification purposes, we reduced the source code of the application to only those commands that contain system calls as illustrated by the following Listing 8.1. All the other commands have been deleted and are denoted by '...' (dots). Also, we are not yet interested into the arguments of the system calls so that they have also been replaced by '...'.

Listing 8.1: Extract of a simple application code example

```
void* p,q;
...
int j = 0; int n = 2;

if (p == NULL) {
  p = malloc(size);  // system call ID '7'
  j++;
}

q = malloc(size); // system call ID '7'

fopen(...);    // system call ID '0'

for (int i = j; i < n; i++){
   ...
   fread(...);    // system call ID '2'
   ...
   fwrite(...);   // system call ID '3'
   ...
}
fclose(...);     // system call ID '1'

if (...)
  printToStdOut(...); // system call ID '26'
...
free();    // system call ID '8'

if(...)
  free(...);   // system call ID '8'
}
```

Every system call is assigned an ID which is denoted in the comment of the according command line. (Up to now, we have addressed system call IDs in an abstract manner in forms of letters such as *A*, *B*, *C*, and *D*. As this example shows a concrete implementation, we now use the real IDs of the system calls as assigned in the system. They can be retraced in the Appendix A.) Considering the control flow of this application example, the following system call sequences are possible to be executed by this application task (of course, all sequences ending with the Sequence Ending Symbol inserted at the task's termination):

Sequence S1 7 7 0 2 3 2 3 1 8 $

Sequence S2 7 0 2 3 2 3 1 8 $

Sequence S3 7 0 1 8 $

Sequence S4 7 0 2 3 1 8 $

Sequence S5 7 0 2 3 2 3 1 26 8 $

Sequence S6 `7 0 2 3 2 3 1 8 8 $`

Obviously, the presented application code may also produce further system call sequences, so that these sequences shall be considered as a sample without any claim to be complete. However, these examples show some variations that are sufficient for further explanations concerning the Behavior Knowledge Base.

Let us assume, that **Sequence S1** is executed by the first instance of the sample task, of course the system calls arriving in a consecutive manner. The Suffix Tree (as up to now being empty) has to be constructed for that sequence by calling:

Call 1 `addNode(&7,false)`

Call 2 `addNode(&7,false)`

Call 3 `addNode(&0,false)`

Call 4 `addNode(&2,false)`

Call 5 `addNode(&3,false)`

Call 6 `addNode(&2,false)`

Call 7 `addNode(&3,false)`

Call 8 `addNode(&1,false)`

Call 9 `addNode(&8,false)`

Call 10 `addNode($,true)`

Figures 8.10 to 8.19 show the successive construction of the Suffix Tree for **Sequence S1**. Figure 8.10 shows the Suffix Tree after the system call ID 7 was inserted by **Call 1**. For any extension by a next symbol of the sequence, we need to know all paths to suffixes of the current sequence as they are candidates to be extended by that next symbol. In fact, the paths of the current suffixes are represented by the last inserted nodes (or at least matching nodes if they have existed in the Suffix Tree before). In our implementation, we use the attribute `pathpool` for the `SuffixTree`-object to store the nodes that are affected in the case of adding the next node. These `pathpool`-nodes are blue-marked in the following figures illustrating the Suffix Tree construction.

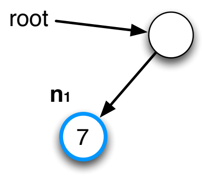


Figure 8.10: Suffix Tree for symbol 7

The next symbol in our system call sequence **Sequence S1** is the system call with ID 7 (at the second position in the sequence). For adding this symbol into the Suffix Tree, **Call 2** is called and the system call sequence executed up to now is $S'_1 = 7\,7$. The objective of the Suffix Tree extension is that afterwards, the Suffix Tree contains all paths to suffixes of the sequences. In this examples, it concerns the suffixes $S'_1[1..2] = 7\,7$ and $S'_1[2..2] = 7$. As a general rule, the algorithm has to extend each node included in the `pathpool` of the former phase $j-1$ and the root node by the currently arriving symbol $S(j)$. Referring to the progress of our example, the `pathpool` consist of n_1 and the current symbol is $S(2) = 7$. But extending the sequence by a symbol in this context does not automatically mean adding of nodes at the nodes stored in the `pathpool` and the root node. It furthermore is concerned with checking whether the `pathpool`-nodes and root already possess a child node with that symbol. The example provided generates that situation: the symbol $S(2) = 7$ that is dedicated to extend the Suffix Tree in phase 2 is equal to the (first) symbol previously included in the Suffix Tree. As n_1 does not has any children, it is extended by the new node n_2 with the symbol $S(2) = 7$. But the root node already has a child with the ID 7. Hence, there is no extension required related to symbol $S(2) = 7$. After inserting the second symbol $S_1(2) = 7$ by **Call 2**, the Suffix Tree for that sequence (illustrated in Figure 8.11) contains paths to all suffixes of the sequence, which means one path to $S'_1[1..2] = 7\,7$ ending in node n_2 and one path to $S'_1[2..2] = 7$ ending in node n_1. Now, the nodes n_2 and n_1 are marked as ending nodes of the suffixes of the present sequence and they are selected into the set of `pathpool` nodes that will be affected by the next symbol extension. (For the following, we will only assign identifiers n_i to those nodes that are essential for the understanding the corresponding explanations.)

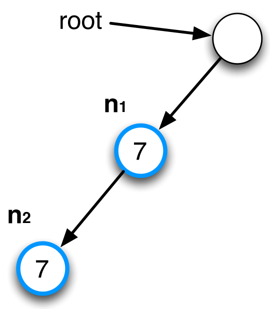


Figure 8.11: Suffix Tree for sequence 7 7

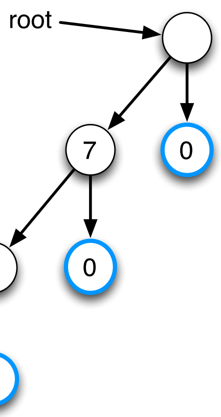


Figure 8.12: Suffix Tree for sequence 7 7 0

Figure 8.12 shows the Suffix Tree after **Call 3** which inserted the system call ID 0. Both previously blue-marked nodes as well as the root node are now extended by a new node containing system call ID 0.

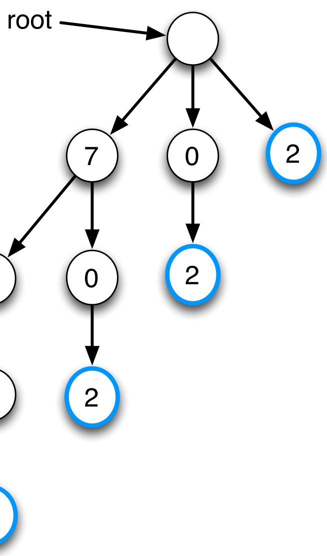


Figure 8.13: Suffix Tree for sequence 7 7 0 2

Figure 8.13 shows the Suffix Tree after inserting the system call ID 2 by **Call 4**. The root node as well as all the previously blue-marked nodes have been extended by a node having the symbol 2.

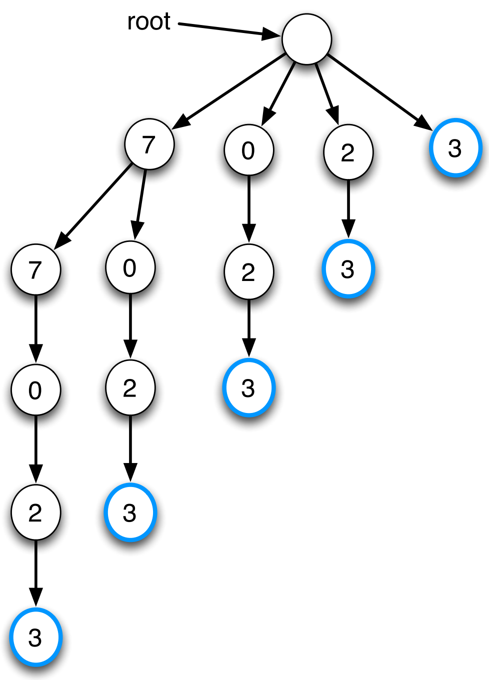


Figure 8.14: Suffix Tree for sequence 7 7 0 2 3

Figure 8.14 shows the Suffix Tree after inserting the system call ID 3 by **Call 5**. The root node as well as all the previously blue-marked nodes have been extended by a node having the symbol 3.

Now, **Call 6** is intended to insert the system call ID 2 as the next symbol of the sequences. Figure 8.15 shows the extension of all previously blue-marked nodes by a node containing the symbol 2 as no children with this ID have existed for the blue-marked `pathpool`-nodes. However, from the root node, a path to a node with ID 2 already exists and is identified by node n_3. Therefore, it does not need to be added, but it is marked blue as a candidate for next symbol extension and included into the `pathpool`. In fact, node n_3 identifies the starting symbol of the suffix of the sequence starting at position 6 ($S[6..j] = 2..?$).

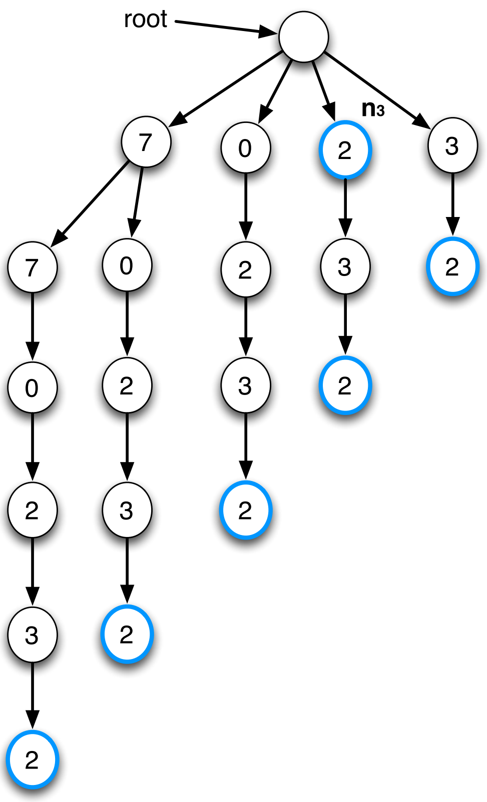


Figure 8.15: Suffix Tree for sequence 7 7 0 2 3 2

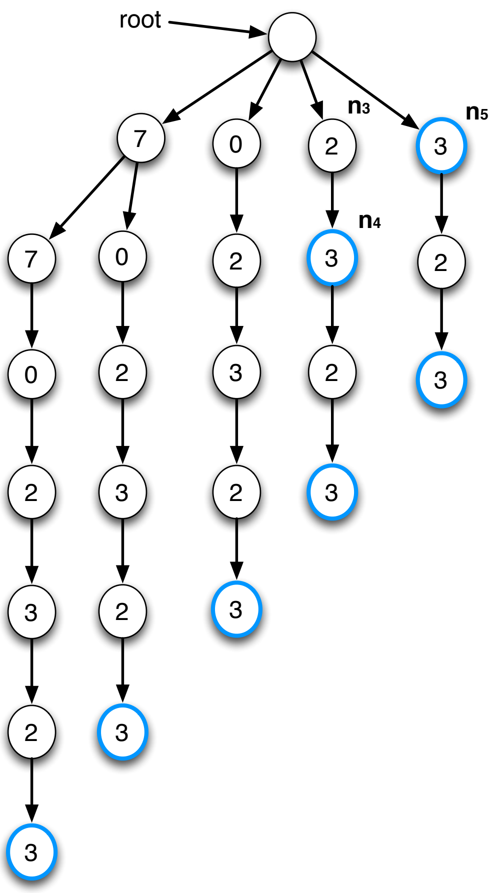


Figure 8.16: Suffix Tree for sequence 7 7 0 2 3 2 3

The next symbol of the sequence is 3 that is inserted by **Call 7**. The resulting Suffix Tree is shown by Figure 8.16. All previously blue-marked nodes, that have been leaf nodes, have been extended by a node containing the symbol 3. These nodes are now marked blue in order to be extended by the next symbol of the sequence. For the node n_3 that determines the suffix starting with 2, no inserting of a new child containing the symbols 3 was required, as it already exists (see node n_4). Hence, only the blue marker is set on that child-node n_4 identifying the symbol 3. Additionally, the node determining the path from root that starts with the symbol 3 is n_5 and, therefore, also marked blue in order to enable to extend suffixes starting with that symbol. After the extension, all the newly inserted nodes as well as those determining the endings of the currently present suffixes are marked blue as they belong to the set of nodes (`pathpool`) affected by the next extension.

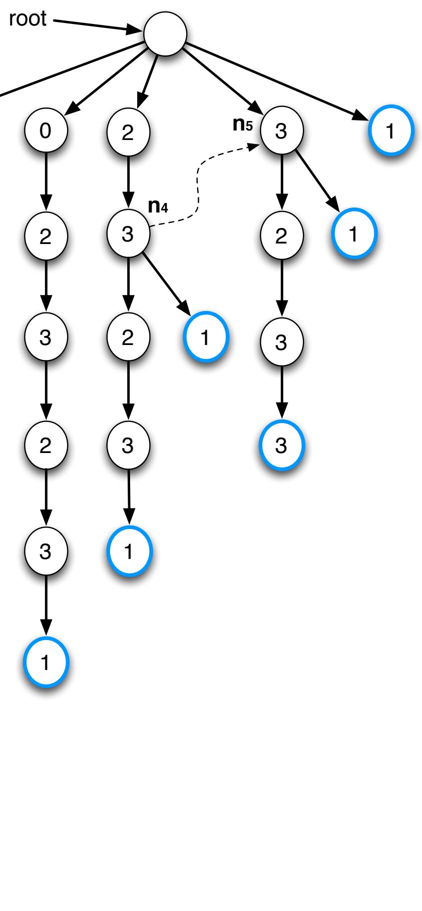


Figure 8.17: Suffix Tree for sequence 7 7 0 2 3 2 3 1

Figure 8.17 shows the Suffix Tree after inserting the system call ID 1 by **Call 8**. All the previously blue-marked leaf node are extended by a new node containing the symbol 1. The remaining previously blue-marked nodes n_4 and n_5 are validated whether child-nodes exist with the symbol 1 exist. As this is not the case, they have to be extended by a new child containing the symbol 1. Now, because these have already had a child-node, they a extended by a further child and therefore, become branch nodes. As they both change their node type into branch nodes within one extension cycle, a suffix links is set between them. Suffix links (see Section 5.2.3) are used to speed up the searching of equal suffixes for extension purposes. In fact, the destination of the suffix link is identifies the path of a suffix that is also a suffix for the subsequence determined by the suffix link source.

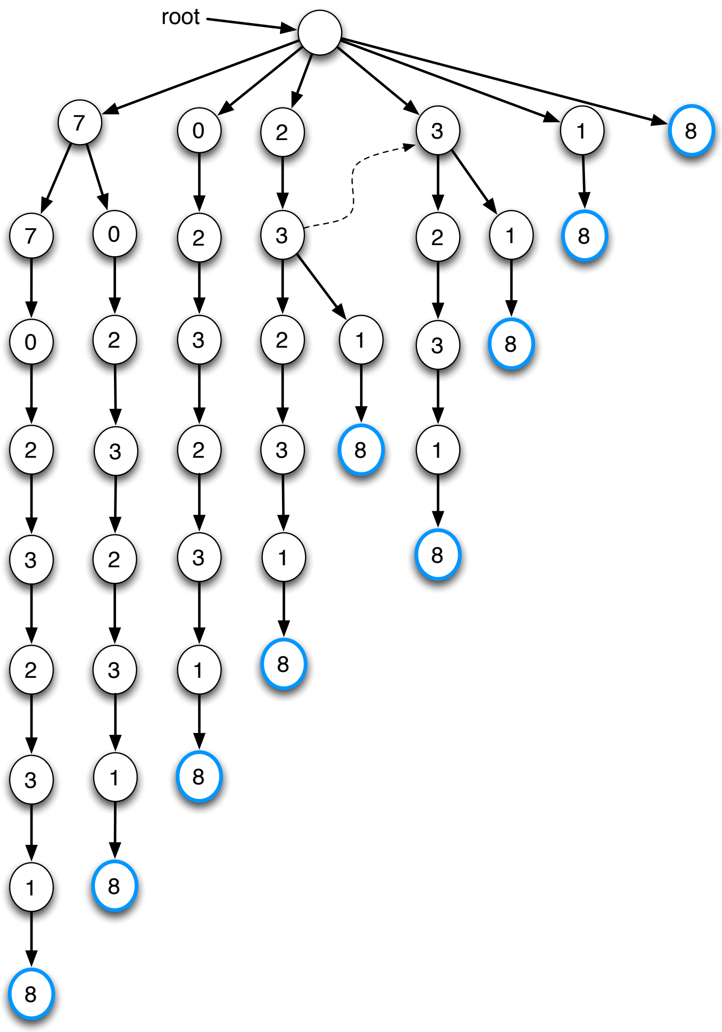


Figure 8.18: Suffix Tree for sequence 7 7 0 2 3 2 3 1 8

Figure 8.18 shows the Suffix Tree after inserting the system call ID 8 by **Call 9**. As all the nodes in the `pathpool` are leaf nodes, the extension is done by simply extending each of the node of the `pathpool` by a new child holding the symbol 8. Additionally, as no path to the symbol 8 has existed before, the root node is also equipped by a node with the symbol 8.

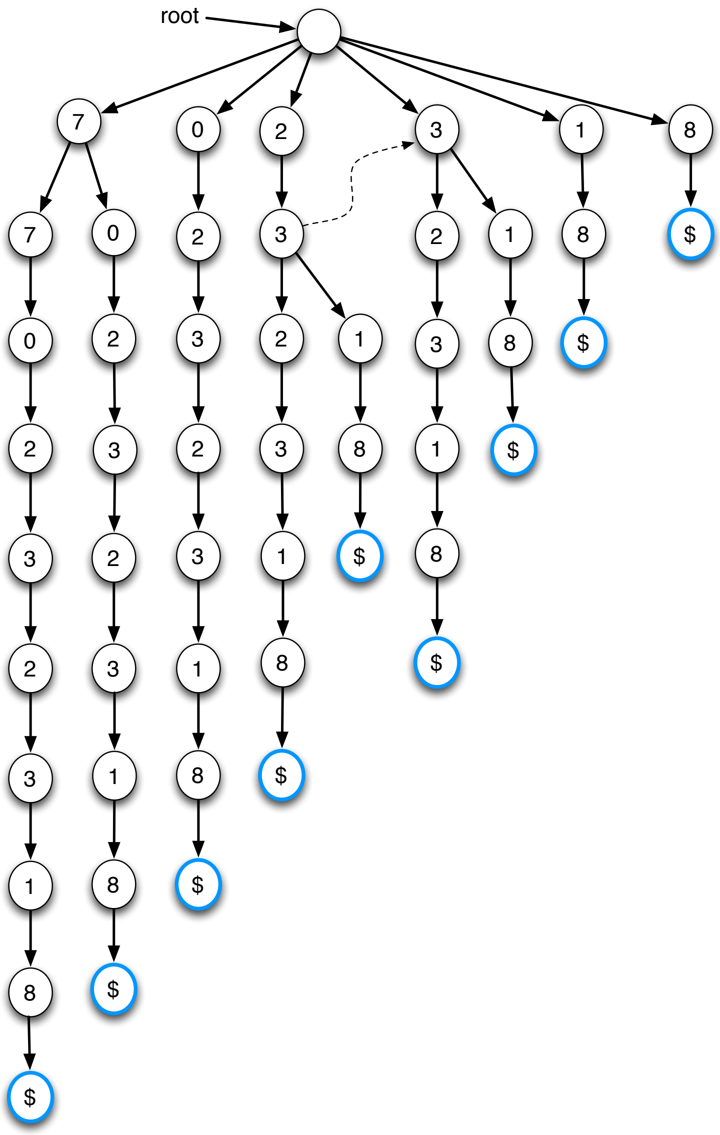


Figure 8.19: Suffix Tree for sequence 7 7 0 2 3 2 3 1 8 $

The last symbol to be included into the Suffix Tree is the Sequence Ending Symbol $. This symbol is included by **Call 10**. As this symbol is the Sequence Ending Symbol, the `endmark` parameter is set to `true`. The extension of the Suffix Tree is processed in the same manner as for any other symbol by extending the set of nodes that are stored in the `pathpool` besides the fact that there is no node with the symbol $ generated that starts from root. This is because of the fact that the Sequence Ending Symbol is determined to identify the ending of a sequence and its according suffixes. It does not make any sense to insert an empty subsequence or suffix into the Behavior Knowledge Base. The final Suffix Tree for the complete behavioral sequence of the first instance of the sample task is now generated and illustrated by Figure 8.19.

8.5.5 Searching and Extension

After the first execution of a task, in fact its first instance, a basis for the Behavior Knowledge Base has been constructed by inserting the behavioral sequence executed by the task's instance into the Suffix Tree. Any further execution of the task requires first the validation whether the newly arising behavioral sequence is already existing in the Suffix Tree or not. Only if the behavioral sequence in not in the Suffix Tree, and extension of the Suffix Tree is required. However, neither the System Call Monitor nor the Behavior Knowledge Base are able to directly recognize subsequences or suffixes of already included sequences. They have to process the newly arriving sequence again in a symbol-by-symbol manner. The validation whether the a path to a symbol of the sequence already exists is already realized in the `addNode`-method as it checks whether from the actually reached node a path to the next symbol to be inserted exists. For each new task instance, the actual reached node is set to the `root`-node as each new behavioral sequence has to be completely inserted into the Suffix Tree which means that its path should start from the `root`-node.

Considering for example the **Sequence S2** (see Section 8.5.4 above) composed of 7 0 2 3 2 3 1 8 $, it is a subsequence of **Sequence S1**. Moreover, it is its suffix starting at the second symbol. Obviously, due to the characteristics of Suffix Trees, each suffix of the sequence is included into the Suffix Tree after its construction. This can be also seen by taking a look at Figure 8.19. Following the path that starts with symbol 7 which is a branch point, and then following to node 0 from the branch point, we will detect **Sequence S2** being in the Suffix Tree. Hence, calling the `addNode`-method for each symbol of that sequence will quickly return after detecting that the path to that symbol already exists.

This will be different for the task instance executing **Sequence S3** = 7 0 1 8 $. This sequence is not a subsequence of the previously included and is therefore not existing in the Suffix Tree yet. After the last task's instance termination, the pointer to the last reached node was set to the `root`-node and the `pathpool` is empty. Hence, at the beginning of a new period, the first symbol of the behavioral sequence 7 is checked against the children of the root node. As that symbol exists in the Suffix Tree, the node n_1 identified by 7 is included into the `pathpool` as a candidate for validating the existence of the next occurring symbol. By this, the path found for the present sequence $S[1..1] = 7$ is followed. Calling `addNode(&0, false)` to add the second symbol of that sequence will detect the next existing path to be followed (in fact, it is a `pathpool` of nodes n_6 and n_7 as the algorithm has to follow the paths of the suffixes starting at $S[1..2] = 7\ 0$ as well as at $S[2..2] = 0$). Now, calling `addNode(&1, false)` detects that the reached nodes in the `pathpool` do not have any children to a node containing the symbol 1. Therefore, these paths have to be extended by newly generated nodes for 1 which is implemented in the `addNode`-method in the same manner as for the construction of the Suffix Tree. Again, the nodes n_6 and n_7 have already had children so that adding nodes with ID 1 makes them become branch nodes. And, as this is executed within one extension cycle, a suffix link is set from node n_6 to node n_7. Because from root, a child-node with symbol 1 already existed having been added by one of the previously executed task instances, no extension of root node is required here. For the last symbol of the sequence, symbol 8, the `addNode`-method is executed according to the process already described. The sequence is then finalized by including the Sequence Ending Symbol at the affected paths. The resulting Suffix Tree for the task having executed **Sequence S1**, **Sequence S2** (which was already contained in the Suffix Tree and therefore needed no extension) and **Sequence S3** (extensions on the Suffix Tree are marked in blue) is shown in the Figure 8.20.

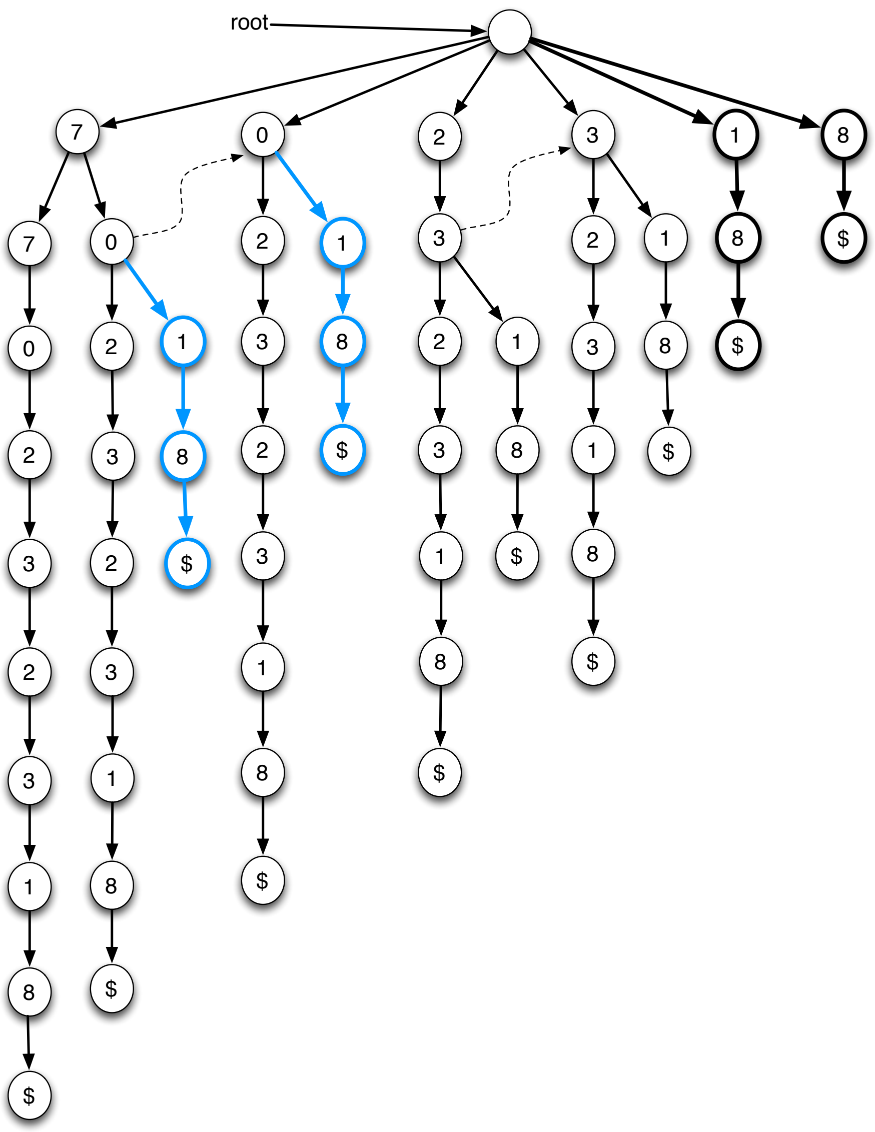


Figure 8.20: Extension of Suffix Tree by sequence 7 0 1 8 $

8.5.6 Discussion on Complexity

As in real-time systems we are facing strong restrictions in terms of time and space (concerning memory), it is necessary to examine the runtime overhead and the memory requirements caused by the Behavior Knowledge Base. This section deals with a static analysis of that problem. Experimental results on the quantitive costs of the Behavior Knowledge Base will be discussed in the Quantitive Evaluation in Section 9.3.

Size of Suffix Tree

Due to the fact, that RTOS have to be fully deterministic, dynamical memory allocation shall be strictly avoided because time for allocation may become unbounded which may lead to unpredictability in terms of time. The memory required by each component shall be known in advance and allocated statically. Hence, this requirement is also set on the Behavior Knowledge Base.

Therefore, when initializing the Suffix Tree for a task, the size of the Suffix Tree is preliminarily configured by the system designer in the offline system configuration (by means of the ORCOS' SCL Configuration see Section 6.2). Here, all parameters such as the maximum number of nodes, the maximal length of paths, the max number of sons of a node, etc. can be configured. Because the application developer is the one that has expert knowledge about the application, he is able to set these parameters according the specific characteristics of the applications.

Nevertheless, static memory allocation means that memory space is allocated in advance, of course always considering worst case requirements. This requires an exhaustive analysis as it is important not to allocate too much memory that will stay unused. To configure the parameters of the Suffix Tree, an analysis of the control flow on the basis of application's system calls is required. Referring to the Section 5.2 (background on Suffix Trees), for an n-length sequence, the according Suffix Tree requires $O(n^2)$ number of symbols if the symbols are assigned labels at the edges. Here, we are interested in a more precise analysis of the size of the Suffix Tree. In our implementation, each symbol of a sequence is represented by one node in the Suffix Tree, therefore, the number of symbols is equal to the number of nodes so that we are arguing on number of nodes in the following. (We are not counting the `root`-node.)

For one sequence of length n, the Suffix Tree contains a path consisting of n nodes for the sequence itself, and paths of $n-1$, $n-2$, ... , $n-(n+1)=1$ nodes for all the suffixes of that sequence. Assuming that all the symbols of the sequence differ (no recurrence of the symbols), each suffix is determined by one unique path in the Suffix Tree. If all the symbols are unique in the sequences ($\forall s_i \in S_k : s_i \neq s_j(i, j = 0..n, i \neq j)$), consequently, the size of the alphabet for that sequence is $m = n$. Then, the Suffix Tree will have no branch nodes and is composed of

$$n + (n-1) + (n-2) + ... + 2 + 1 = \sum_{i=1}^{n} i = \frac{n^2+n}{2} = \frac{n(n+1)}{2} \tag{8.1}$$

nodes.

Each suffix (the sequence itself is also a suffix of itself) is ending with a node containing the Sequence Ending Symbol \$ (except the empty suffix) which in fact leads to n further nodes. (We do not add the \$ as a son of the `root`-node into the Suffix Tree.) Hence, for a sequence of length n with n different symbols, the sequence becomes the length $n' = n + 1$ when extended by the Sequence Ending Symbol \$. The resulting Suffix Tree consists of

$$\sum_{i=1}^{n} i + n = \frac{n^2 + n}{2} + n = \frac{n(n+1)}{2} + n = \frac{n'(n'+1)}{2} - 1 \tag{8.2}$$

nodes.

Considering the **Sequence S3** = 7 0 1 8 \$, for example, the Suffix Tree is shown in Figure 8.21. **Sequence S3** has the length $n = 4$ and $n' = 5$, and consists of $m = 4$ different symbols. The Suffix Tree consists of $\frac{n(n+1)}{2} + n = \frac{4\cdot(4+1)}{2} + 4 = 14$ nodes (ignoring the `root`-node). We come to the same result by using $\frac{n'(n'+1)}{2} - 1 = \frac{5(5+1)}{2} - 1 = 14$. When counting the nodes in the Figure 8.21, we will see that the result of 14 nodes is correct.

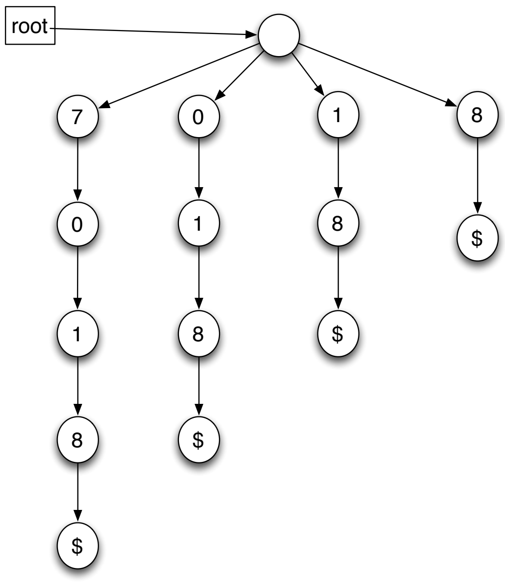


Figure 8.21: Suffix Tree for sequence 7 0 1 8 \$

Assuming that all symbols are different within one sequence ($m = n$) points out a special case. Now, let us consider the number of nodes for sequences with recurring symbols. In fact, the size m of the set of symbols used must be smaller then the length of the sequence n so that $m < n$. For sequences that contain multiple occurrences of the same symbol, the Suffix Tree does not contain one separate path for each suffix of a sequence any more. Moreover, for a suffix that starts with a symbol already occurred in the prefix of the sequence, a node with that symbol is already existing in the Suffix Tree. However, it is not enough to only count the occurrence of a symbol in the sequences. To obtain the accurate size of a Suffix Tree, we have to consider whether there are exactly matching subsequences in a sequence. Subsequence in this context has the minimum length of 1. Multiple (at least 2) occurrences of subsequences within one sequence share one path in the Suffix Tree for the length of that subsequence. Then, the previously determined number of nodes in the Suffix Tree (see Equation 8.4) has to reduced by the length of each matching subsequence (as they share the same path).

For each suffix of a sequence, it is essential to verify whether the prefix of that suffix is equal to a subsequence already occurred in the prefix of that sequence. A matching subsequence will start at the first symbol of the suffix and may go until the end of the sequence. (If that subsequence is not ending at the last symbol of the sequence, the path of the subsequence in the Suffix Tree ends at a branch node.)

For each suffix of the sequence the length of matching (already occurred) subsequences is determined by the following algorithm:

Algorithm Counting the length of recurring subsequences within one sequence

```
Require: Behavior sequence $S_k$ of a task

  $numOfReducingNodes \leftarrow 0$
  for $i \leftarrow 2, n$ do

      for $j \leftarrow 1, (i-1)$ do
          while $S_k[j] = S_k[i]$ and $i \leq n'$ do
              $numOfReducingNodes \leftarrow numOfReducingNodes + 1$
              $j \leftarrow j + 1$
              $i \leftarrow i + 1$
          end while
      end for
      end for

  return $numOfReducingNodes$: number of nodes to reduce the size of the Suffix Tree
```

Then the resulting size of the Suffix Tree is defined by

$$|SuffixTree_T| = \frac{n(n+1)}{2} + n - numOfReducingNodes \qquad (8.3)$$

with $numOfReducingNodes = \sum_{i=1}^{k} length(A_i)$ with A_i determining a subsequence occurring in the prefix of the sequence being equal to the prefix of a suffix, and k the number of that subsequences found in the sequence.

To explain this more precisely, we will use the example for **Sequence S1** (7 7 0 2 3 2 3 1 8 \$) from the section above. By using the algorithm presented above, we can extract the following subsequences as prefixes of the considered suffix that have already occurred in the prefix of the sequence:

1. $A_1 = \{7\}$ with length 1 for $i = 2$ and $j = 1$
2. $A_2 = \{2\ 3\}$ with length 2 for $i = 6..7$ and $j = 4..5$
3. $A_3 = \{3\}$ with length 1 for $i = 7$ and $j = 5$

Hence, the size of the Suffix Tree calculated by Equation 8.4 has to be reduced by the sum of the length of extracted subsequences $1+2+1=4$ ($n=9$ for that sequence) which leads to:

$$\frac{n(n+1)}{2}+n-\sum_{i=1}^{k} length(T_i) =$$

$$\frac{9(9+1)}{2}+9-\sum_{i=1}^{3} length(T_i) =$$

$$45+9-(length(T_1)+length(T_2)+(length(T_3)) =$$

$$54-(1+2+1)=50$$

Resulting from this, the Suffix Tree for **Sequence S1** has 50 nodes which is correct when taking a look at Figure 8.19.

In reality, it is not possible to predict the number and the length of recurring subsequences. But if m_k - the size of the alphabet of a sequence S_k - is $m_k < n_k$ (n_k length of the sequence S_k), the it is obvious that exactly $n_k - m_k$ symbol reoccur in the sequence without taking into account whether they belong to reoccurring subsequence. Hence, the worst estimation for the number of nodes in a Suffix Tree for one sequence is:

$$|SuffixTree_T| = \frac{n(n+1)}{2}+n-(n-m) = \frac{n(n+1)}{2}+m \tag{8.4}$$

This equation is also valid for sequences with $m=n$ and formulates the general upper bound for the Suffix Tree size build up for one sequence.

However, the Suffix Tree implementation in intended not only to hold one sequence but to hold all the behavior sequences that a task has executed. Therefore, it is essential to analyze the size of a Suffix Tree containing multiple behavior sequences.

First, let us consider the theoretically possible size of a Suffix Tree. For each task, a maximal length of a behavioral sequence n_{max} can be determined by analyzing the control flow of a task and (the worst case from the viewpoint of our anomaly detection) is defined by the path in the task's control flow that produces the maximum number of system calls. ORCOS actually offers $m_{max}=73$ different system calls (see Appendix A). For a n_{max}-length sequence, hence we get $m_{max}^{n_{max}}$ permutations as each system call can occur at any position in the behavioral sequence. Furthermore, as n_{max} declared the worst case, all sequences of length from $1, 2, ..., n_{max}-1$ are also possible. Therefore,

$$m_{max}^{n_{max}} + m_{max}^{n_{max}-1} + ... + m_{max}^{2} + m_{max}^{1} = \sum_{i=1}^{n_{max}} m_{max}^{i} \tag{8.5}$$

defines the number of nodes for all possible permutations of sequences with maximal length of n_{max}. Taking into account, that each sequence is extended by the Sequence Ending Symbol \$, each node in the Suffix Tree will be extended by one further son for the symbol \$, we get

$$|SuffixTree_T| = 2\cdot\sum_{i=1}^{n_{max}} m_{max}^{i} \tag{8.6}$$

For $m_{max}=73$, as Equation8.6 is an exponential function, the size of the Suffix Tree (in theory) becomes really huge.

Now, from practical point of view, we can reduce this size enormously:

1. it is very improbable that one task uses all $m_{max} = 73$ system calls. Usually, a task T executes only a small subset of system calls which results in $m_T \lll m_{max}$, and therefore

   $$2 \cdot \sum_{i=1}^{n_{max}} m_T^i \lll 2 \cdot \sum_{i=1}^{n_{max}} m_{max}^i \tag{8.7}$$

   the theoretically possible size becomes much smaller.

2. there are dependencies between the system calls that have to be respected when programming the application task. For example, considering the memory related system calls (all system calls are listed in Appendix A), each call of the system call `malloc` (ID 7) requires the call of `free` (ID 8) at a later point of time. If the application is programmed correctly, a `free` should never be called if no `malloc` has been executed before.

   Another example for a dependency is provided by the socket related system calls. A `listen` (ID 21) to a socket leads to a runtime error if the socket was never created before, by calling `socket` (ID 19). There are numerous of those examples showing the dependencies between single system calls. Sticking to that dependencies, only *legal* sequences shall be executed by tasks and hence, only *legal* sequences shall be inserted into the Suffix Tree which again reduces the theoretic size of a Suffix Tree.

   However, our anomaly detection is intended to also detect executions of *illegal* system call sequences. Therefore, they cannot be completely ignored by the Suffix Tree. We will explain what happens if a task executes such an unstable or *illegal* system call sequence in more detail in Section 8.7.

3. We expect the particular executions of task instances to be *similar*. Similar in that context means similar in terms of the system call sequences the task instances execute as they belong to one task that, in fact, has one unique implementation. Hence, we expect many recurring subsequences in the behavior sequences of a task. Mainly, based on this assumption, we have proposed to use Suffix Trees for Behavior Knowledge Base. As recurring subsequences share the same paths in the Suffix Tree (they are not included for multiple times), the size of the Suffix Tree is predominantly determined by the variation of subsequences that compose the behavioral sequences.

The latter aspect, is the key ingredient for determining the concrete size of the Suffix Tree containing a task's behavioral sequences. For the first instance, the size of the Suffix Tree is defined by Equation 8.4 $|SuffixTree_{T_1}| = \frac{n_1(n_1+1)}{2} + m_1$. For the second instance, the Suffix Tree has to be evaluated whether it already contains the sequences and/or has to be extended.

Now, we would like to provide a mathematical upper bound for the size of Suffix Tree containing multiple sequences. In fact, we can say, that for each sequence T_i an own Suffix Tree has to be constructed having the size of $\frac{n_i(n_i+1)}{2} + m_i$ and is then merged with the already existing Suffix Tree T_{i-1}. The nodes that are existing in both Suffix Trees before merging are the ones that do not have to be added to the Suffix Tree T_{i-1} to obtain Suffix Tree T_i so that they reduce the number of added nodes. It is strongly related to the set of symbols that the single behavioral sequences are composed of. Let us define M_i as the set of symbols for a sequence S_i of the task T by $M_i = \{s_1, s_2, ..., s_z\}$ with $|M_i| = m_i = z$ ($s_1..s_z$ being the symbols or system call IDs), and M_T the (complete) set of symbols of the task T. M_T is the union of all the set of symbols over all (l) behavioral sequences of the task: $M_T = \bigcup_{j=1}^{l} M_j$

When a novel behavioral sequence occurs, we can distinguish between 3 different cases:

1. The symbols that sequence S_i is composed of have not occurred in the previous sequences of the task and are therefor not existing as nodes in the Suffix Tree. This means that the previously occurred system calls are united to $M_{T_{i-1}} = \bigcup_{j=1}^{i-1} M_j$ and hence, as all system calls of S_i have not occurred before: $|M_{T_{i-1}} \cap M_i| = 0$. Then, the number of nodes of the united Suffix Tree T_i is

$$|SuffixTree_{T_i}| = |SuffixTree_{T_{i-1}}| + |SuffixTree_{S_i}| = |SuffixTree_{T_{i-1}}| + \frac{n_i(n_i+1)}{2} + m_i$$

   This case is possible as long as $|M_{T_{i-1}} \leq << m_{max} - m_i$

2. The symbols that sequence S_i is composed of have all already occurred in the previous sequences of the task and are therefor all included as nodes in the Suffix Tree: $M_{T_{i-1}} = \bigcup_{j=1}^{i-1} M_j = \bigcup_{j=1}^{i} M_j = M_{T_{i-1}}$ and hence $|M_T \cap M_i| = |M_i| = m_i$. Then, in the Suffix Tree, at least one node for each symbol exists as a son of the `root`- node. Hence, without knowing whether there are matching subsequences, the size of the Suffix Tree is bounded by

$$|SuffixTree_{T_i}| = |SuffixTree_{T_i-1}| + |SuffixTree_{S_i}| - m_i$$

$$= |SuffixTree_{T_{i-1}}| + \frac{n_i(n_i+1)}{2} + m_i - m_i = |SuffixTree_{T_{i-1}}| + \frac{n_i(n_i+1)}{2}$$

3. Some of the symbols that sequence S_i is composed of have all already occurred in the previous sequences of the task and therefore, those symbols are included as nodes in the Suffix Tree: $M_{T_{i-1}} = \bigcup_{j=1}^{i-1} M_j$ and hence $|M_{T_{i-1}} \cap M_i| = |M_i| = r$ with $1 < r < m_i$ Then in the Suffix Tree, at least r nodes for the matching symbol exist as a sons of the `root`- node. In that case, the size of the Suffix Tree is bounded by

$$|SuffixTree_{T_i}| = |SuffixTree_{T_{i-1}}| + |SuffixTree_{S_i}| - r$$

All these 3 cases have a common structure consisting of

$$|SuffixTree_{T_i}| = |SuffixTree_{T_{i-1}}| + |SuffixTree_{S_i}| - x$$

with $x = 0 = |M_{T_{i-1}} \cap M_i|$ for case 1, $x = m_i = |M_{T_{i-1}} \cap M_i|$ for case 2, and $x = r = |M_{T_{i-1}} \cap M_i|$ resulting into:

$$|SuffixTree_{T_i}| = |SuffixTree_{T_{i-1}}| + |SuffixTree_{S_i}| - |M_{T_{i-1}} \cap M_i|$$

For defining the upper bound for the size of the Suffix Tree T, we have to sum up over all sequences and their according Suffix Trees. Based on this, we can generally formulate:

$$|SuffixTree_T| = \sum_{k=1}^{l} \left(\frac{n_k(n_k+1)}{2} + |M_k| - |\bigcup_{j=1}^{k-1} M_j \cap M_k| \right) \tag{8.8}$$

which in worst case without knowing the particular length of the sequences results in

$$|SuffixTree_T| = \sum_{k=1}^{l} \frac{n_{max}(n_{max}+1)}{2} + \sum_{k=1}^{l} \left(|M_k| - |\bigcup_{j=1}^{k-1} M_j \cap M_k| \right) \tag{8.9}$$

The latter sum is bounded on the maximum number of system calls used by the task and results in

$$|SuffixTree_T| = l \cdot \frac{n_{max}(n_{max}+1)}{2} + |M_T| \tag{8.10}$$

In (the theoretic) worst case, meaning that there are all permutations of system call possible to be executed by the task, this upper bound may reach exponential size of $2 \cdot \sum_{i=1}^{n_{max}} m_T^i$. Contrary, if for each sequence S_i it is valid that it is a *real* sub-sequence of already occurred sequences (or equal to it) then we declare this as the best case and the size of the resulting Suffix Tree is

$$|SuffixTree_{T_i}| = |SuffixTree_{T_i-1} = |SuffixTree_{T_{max}}| = \frac{n_{max}(n_{max}+1)}{2} + |M_T|$$

The real size of the Suffix Tree, however, can only be determined when knowing all sequences. Assuming that, we are able to calculate the concrete size. The following algorithm illustrates the procedure to calculate the number of nodes that have already existed in the Suffix Tree and do not have to be extended when inserting a new sequence S_i(it is similar to the one presented above for a Suffix Tree containing one sequence).

Hence, the number of nodes added by one further sequence S_i is the number of nodes that would be required for a Suffix Tree for that sequence minus the *numberOfReducingNodes* that already exist in the Suffix Tree:

$$\#numOfNodes_i = |SuffixTree_{T_i}| - numOfReducingNodes_i \tag{8.11}$$

For a number of i sequences, the Suffix Tree size can be calculated by

$$|SuffixTree_{T_i}| = |SuffixTree_{T_1}| + (|SuffixTree_{T_2}| - numOfReducingNodes_2) + ...$$

$$... + (|SuffixTree_{T_i}| - numOfReducingNodes_i)$$

$$= \sum_{j=1}^{i} \left(|SuffixTree_{T_j}| - numOfReducingNodes_j \right)$$

$$= \sum_{j=1}^{i} \left(\frac{n_j(n_j+1)}{2} + m_j - numOfReducingNodes_j \right) \tag{8.12}$$

This can, of course, only be calculated by checking the number of already existing nodes for each of the sequences. It requires to analyze each sequence in detail to obtain the concrete number of nodes for Suffix Tree T_i.

Algorithm Counting the number of nodes already existing in the Suffix Tree when inserting a new sequence

Require: Suffix Tree of a task, Behavior sequence S_k of a task

```
TreeNode* $actualnode \leftarrow rootnode$
int $numOfReducingNodes \leftarrow 0$
for $i \leftarrow 1, n_k$ do

    TreeNode* $temp \leftarrow getNodeByValue(S_k[i], actualnode)$
    $j \leftarrow i + 1$

    while $temp$ and $j \leq n'_k$ do
        $actualnode \leftarrow temp$
        $temp \leftarrow getNodeByValue(S_k[j], actualnode)$
        $j \leftarrow j + 1$
    end while

    $numOfReducingNodes \leftarrow numOfReducingNodes + j - i$
    $actualnode \leftarrow rootnode$

end for

return $numOfReducingNodes$: number of nodes that are already inserted in the present Suffix Tree
```

Nevertheless, we expect the sequences to be *similar* and to consist of a bounded set of subsequences determined by the control flow of the applications task. Hence, we expect the real size of the Suffix Tree for a task to be closer to $|SuffixTree_T| = \frac{n_{max}(n_{max}+1)}{2} + m_T$ than to the worst case.

Referring to the example of the previous Section 8.5.5, the Suffix Tree T_1 has been constructed for **Sequence S1** = 7 7 0 2 3 2 3 1 8 $and now was extended by **Sequence S3** = 7 0 1 8 $ (see Figure 8.20). For **Sequence S3** ($n_3 = 4$, $m_3 = 4$) the size of the Suffix Tree would be $\frac{n_3(n_3+1)}{2} + m_3 = \frac{4 \cdot 5}{2} + 4 = 14$ and the number of already existing nodes is calculated by the algorithm presented above that leads to $numOfReducingNodes = 2 + 1 + 3 + 2 = 8$ as for the suffixes the following subsequences match the already existing Suffix Tree:

1. $A_1 = \{\ 7\ 0\ \}$ with length 2
2. $A_2 = \{\ 0\ \}$ with length 1
3. $A_3 = \{\ 1\ 8\ \$\ \}$ with length 3 and
4. $A_4 = \{\ 8\ \$\ \}$ with length 2

Hence, the number of nodes the Suffix Tree T_1 has to be extended when inserting **Sequence S3** is $\frac{n_3(n_3+1)}{2} + m_3 - numOfReducingNodes = 14 - 8 = 6$ which is correct when comparing that with Figure 8.20. For the **Sequence S2** = 7 0 2 3 2 3 1 8 $, $numOfReducingNodes = \frac{n_2(n_2+1)}{2} + m_2$

and hence, the Suffix Tree T_1 has not to be extended by any node, which is obvious as **Sequence S2** is a subsequence of **Sequence S3**.

Of course, the exact calculation of the size of the Suffix Tree is only possible if all sequences that are executed by a task are known in advance. Contrary, as our anomaly detection is designed for self-reconfiguring applications, the behavior of tasks are unknown (but the set of used system calls is known). However, the presented calculation allows a well-defined estimation of the Suffix Tree (maximum) size required for an application. Experimental results of Suffix Tree size are provided in Section 9.3.

Analysis of Runtime Overhead

Runtime overhead is produced in each period of a task as each task instance generate one behavior sequence that has to be inserted into the Behavior Knowledge Base. Moreover, runtime overhead is produced by each system call executed by the task.

Any executed system call is a symbol in the behavior sequence that is, first, checked whether it is contained in the current path in the Suffix Tree, and, if not, inserted as a node into the Suffix Tree. The matching process is always executed and requires constant time. If the current system call is contained in the current path of the Suffix Tree, we can assume that the associated node exists for all suffixes due to the nature of Suffix Tree construction method. The resulting runtime effort remains constant. If no node corresponding to the current system call exist in the current path of the Suffix Tree, it has to be generated and extended to that path as well as all paths determining the suffixes of the behavior sequence. The generation of one node is considered to require constant time. Hence, the runtime overhead for extending the Suffix Tree by a system call of the behavior sequence depends on the number of nodes that are generated.

Every node represents the occurrence of a system call in a suffix of the behavior sequence. The number of occurrences of a system call in the Suffix Tree depends on the number of occurrences of the system call in the suffixes of the corresponding sequence $S = s_1 s_2 ... s_n$. The first symbol in the sequence occurs only in the suffix $S[1..n]$, while the second symbol s_2 is present in the suffix $S[1..n]$ and $S[2..n]$. Generally, each symbol s_i is present in i suffixes from $S[1..n]$ to $S[i..n]$. From this follows that the worst case overhead for extending the Suffix Tree by one system call is depending on the position i of the system call within the behavior sequence S as every system call s_i requires i nodes in i paths to be generated.

Consequently, the entire runtime overhead related to a complete behavior sequence is $O(n^2)$. This runtime overhead has to be taken into account when performing the Schedulability Analysis for a task. By knowing the maximum number of system calls of a task n_{max} - which is the worst case length of a task's behavior sequence -, this runtime overhead is bounded to $O(n_{max}^2)$ for each period of a task.

8.6 Operating System Health Monitor

The proposed anomaly detection approach was inspired by the Danger Theory. The main contribution of the Danger Theory to our anomaly detection is that behavior or observation information is classified based on predominate signals that reflect a health state. In the former parts of this chapter we have introduced the System Call Monitor as the component for collecting behavioral information, and the Behavior Knowledge Base implemented in forms of Suffix Trees as the data structure to represent behavioral sequences and hold the entire behavioral history. However, these behavioral sequences have not been classified yet. In order to build the fundament for the classification of the collected behavioral sequences, signals reflecting the system health state (the respective input signals from Danger Theory) are required. We define the health state of the system by the state of the operating system and its associated components. To supply the classification process with signals reflecting the state of the RTOS, we have integrated an OS Health Monitor into ORCOS.

8.6.1 Design of OS Health Monitor

The basic idea of the OS Health Monitor is to extract parameters that act as software sensors to determine the operating system state while tasks are executing, and to identify whether the operating system parameters are in a healthy, suspicious, or malicious range. We call the parameters that determine the system state as *health data*. (They will be defined in the following sections.) The health data is output by the OS Health Monitor and, consequently, represents the input signals for the classification of the anomaly detection module.

Referring to the basic idea, the OS Health Monitor has to realize the following functionalities:

1. Extracting health data parameters of OS (and its components)
2. Storing and organizing health data (including its history)
3. Reporting the health status of the OS (and its components) to the anomaly detection module

From these functionalities, requirements can be derived resulting into the following design decisions:

1. In order to be able to efficiently access internal status parameters of the operating system, the OS Health Monitor must become a part of ORCOS itself and was therefore integrated into the ORCOS kernel.
2. The system health state cannot be associated with one single attribute of the operating system. The operating system consists of a number of components, the kernel modules, that offer diverse ingredients to reflect their individual states. As these ingredients are represented by parameters that are related to the OS kernel modules, the OS Health Monitor must provide mechanisms to extract the health data from the respective OS components.

   This requirement leads to two design decisions:

   2.1 The attributes that contribute to the overall system health state are specific to each OS component. To be able to answer the question which attributes associated with an OS component are relevant and define the system's *health data*, first, each of the OS components has to be individually analyzed in detail (by experts for that component).

   2.2 Values of parameters that define the *health data* have to be gathered from each OS component (or kernel module respectively). This data is specific to that component, potentially composed of different types, and often encapsulated internally within the OS kernel module not directly accessible from outside the component. Therefore, a component-specific implementation is required for the extraction of the data which is realized by individual OS Health Monitor modules associated with OS components. Then, a central module is responsible for managing the single component monitor modules, on the one hand, and for managing the collected health data, on the other hand. Therefore, an overall hierarchical architecture for the OS Health Monitor was established consisting of monitor modules related to the OS components and the Health Monitor Center.

3. The data collected by the OS Health Monitor are strongly related to the runtime status of the system and its components as it reflects the *health status* of the system at the current time stamp. Consequently, the OS Health Monitor must be a runtime component integrated into the operating systems.

4. The collected health data deals as input signals for the classification of behavior performed by the Anomaly Detection Module. However, values of parameters at a particular time stamp are not the only indicators for classification but in some cases, their evolvement considering the recent history may be of interest. Therefore, apart from current data provided by the monitor modules, history data must also be preserved to some extent by the OS Health Monitor. In order to support to the Anomaly Detection Module with appropriate data, the data gathered by the OS Health Monitor is stored and organized in an OS Health Monitor Database.

5. The OS Health Monitor is responsible for delivering the input signal for the classification of the behavior. These classification input signals are of three types defining healthy, suspicious, or malicious system state. These three states are assigned to different colors, according to the traffic light principle:

   - green - safe signal, healthy state
   - yellow - warning signal, suspicious state
   - red - danger signal, malicious state

   Depending on the collected health data values, the OS Health Monitor produces the according output by means of a Signal Generator.

6. To provide access to the OS Health Monitor functions as well as its data, the OS Health Monitor offers an interface in form of a clearly defined API.

7. Being integrated into ORCOS that is fully configurable offline at module level, the OS Health Monitor is configurable as well. To fully comply with the concepts of ORCOS, the OS Health Monitor is an operating system module that can be activated or deactivated. But it is not only the OS Health Monitor as a unique unit that can be included or excluded into the operating system. Each of the monitor modules can be integrated individually in dependence of the kernel modules configured at design time. These monitor modules can be individually configured as well.

   In particular, the health signals that the OS Health Monitor generates rely on expert knowledge that is required to define boundaries between healthy, suspicious, and malicious state for the affected parameters. In order to supply a user-friendly interface for the experts, the setting of the parameter thresholds will be offered within the ORCOS' configuration tool SCL Editor.

   Domain experts may identify further composition of parameters and relationships between them that may be ingredients for the health state of the OS. These parameter compositions should be validated by the OS Health Monitor even though their validation has not been implemented within the modules. We have developed a Scenario Condition

Framework to allow experts to extend the health parameters, configure relationships between them, define and add new conditions and to make it possible to flexibly integrate them into the OS Health Monitor.

All these design decisions and requirements have been realized by the OS Health Monitor and will be discussed in detail in the following sections.

8.6.2 Integration into ORCOS Architecture

The OS Health Monitor is strongly related to the kernel as it collects data extracted from the kernel modules. Furthermore, the OS Health Monitor' s task is to deliver signals to the Anomaly Detection Module that is responsible for the classification of the behavior. Based on its purpose, the OS Health Monitor forms an interface between the OS components and the Anomaly Detection Module. Figure 8.22 shows the ORCOS architecture with the OS Health Monitor.

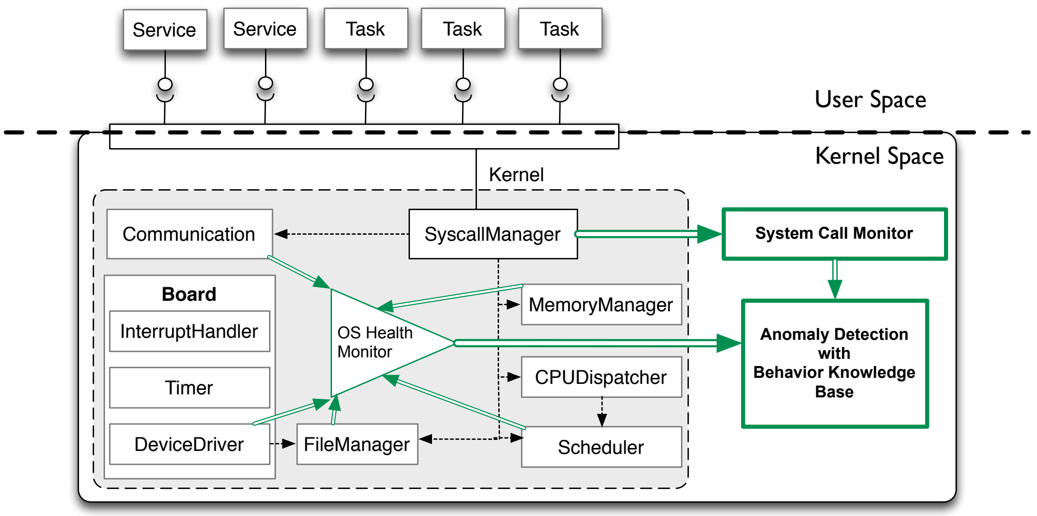


Figure 8.22: Integration of OS Health Monitor into ORCOS Architecture (see Fig. 6.1)

Hierarchical Architecture

The diverse tasks of the OS Health Monitor form enclosed functions that can be encapsulated within separate monitor modules. Therefore, the OS Health Monitor was constructed in a module-based manner:

Component Monitor Modules: for each kernel module, a kernel module-specific monitor has been developed to gather the module-related health parameters.

Monitor Database: the data collected by the component monitor modules is organized and stored in the Monitor Database.

Signal Generator: The health signal is computed based on the data collected and present in the Monitor Database. The Signal Generator implements and proceeds the generation of the Health Monitor's output signal.

A central component is required to manage these modules, their processing and the data flow between them, and to provide interfaces to the Anomaly Detection Module. This task is addressed to the Health Monitor Center.

With that central module, all in all, the OS Health Monitor architecture is organized in a hierarchical manner (see Figure 8.23) consisting of a central module for management, containing the database module and the signal generator, and component monitors that are assigned to each ORCOS kernel module.

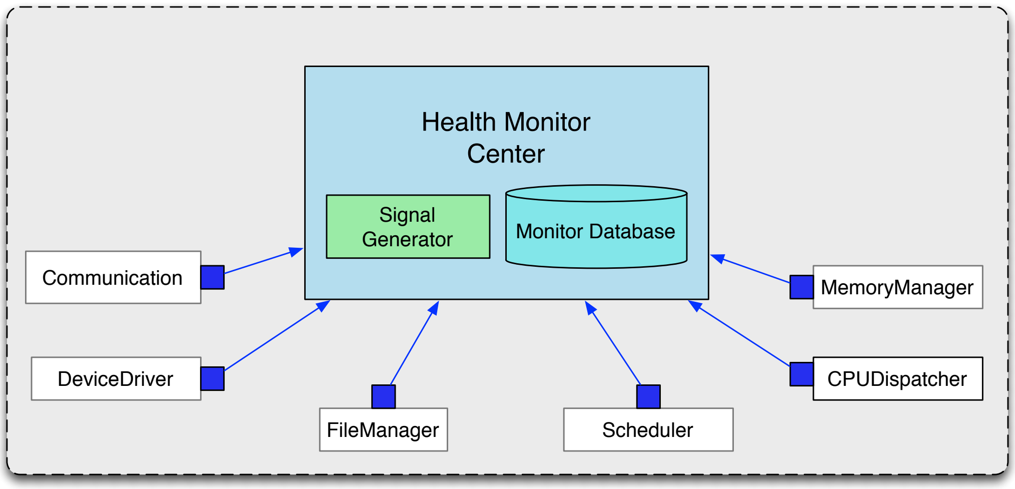


Figure 8.23: Architecture of OS Health Monitor

8.6.3 Component Monitor Modules

The major objective of the OS Health Monitor is to extract the operating system health data. This data is spread over the single components of the operating system which, in ORCOS, are represented by kernel modules. To extract the component-specific data, each kernel module is assigned on component-specific monitor implementation in forms of monitor modules. Therefore, the OS Health Monitor provides the following monitor modules already illustrated in Figure 8.23:

- Scheduler Monitor
- Processor Usage Monitor
- Memory Monitor
- Communication Monitor
- File Manager Monitor
- Device Driver Monitor

In general, the process of data extraction and collection is confronted with three different situations:

1. The respective kernel module offers an interface to directly access the data that has been identified as being a parameter of *health data*. The Monitor Module can gather this data by use of the offered interface.

2. The parameters identified as *health data* can only be accessed internally within the kernel module. Therefore, no interface function is offered for outside access. Then, the Component Monitor Module can only collect that data by directly inserting code into the OS component to extract the according data.

3. Some parameters of *health data* are not directly provided within the OS component. They can only be determined by calculations or accumulations on the basis of the present internal OS component data. The Component Monitor Module has to implement these calculations for the respective health parameters.

Health data that is collected by the OS Health Monitor without any need of being previously processed on the basis of other *health data values* and, that therefore is directly available through interfaces of kernel modules or code extensions, is termed *base health parameter*. Another subset of health parameters are those that represent static values being initialized at the system's or the task's initialization phase. We call this subset *static health parameters*.

Each of the components has been examined in detail to concretely define the parameters to be collected by each Monitor Module. The main aspects of each Monitor Module are described in the later part of this section. The complete listing of the parameters collected by the Component Monitor Modules can be found in Appendix C. (*Base health parameters* are highlighted in boldface.)

The parameters of *health data* have been individually determined for each OS component assisted by expert knowledge:

General Monitor Parameters

In order to identify any unexpected system behavior or inside the OS Health Monitor itself, all Component Monitor Modules measure the time interval in between their own execution. The value is stored in the health parameter `Time Interval`. Each Component Monitor Module stores and updates the `Time Interval` values, and calculates one average (activation) value since system startup and one for the recent short-term history. Indications for unstable behavior may be obtained by comparing the current value for `Time Interval` and its average value. E.g. if a Monitor is called much more often or unusually infrequent than *normally* with respect to the execution frequency in the system's lifetime, the current value for `Time Interval` and the `Average Short-term Time Interval` will strongly deviate from the `Average Longterm Time Interval`. Obviously, these situations may happen in real-life systems. But, such a deviation may also be an indicator for a suspicious system situation, as in real-time systems, any system component shall execute according to a predefined period.

Scheduler Monitor

The Scheduler is responsible for executing the scheduling strategy deciding which task to be executed next. In RTOS, the Scheduler acts on the basis of tasks' priorities in forms of deadlines, execution time etc. Consequently, the Scheduler has all the execution-related and scheduling-related data of each task. In ORCOS, the Scheduler is executed whenever the according timer interrupt is raised as timer interrupts identify the time stamps for context switches.

The Scheduler Monitor is intended to determine the health state of the Scheduler. Therefore, it has to differentiate between stable and unstable behavior of the Scheduler itself as well as of the scheduled tasks. Even though the operating system supports different types of scheduling strategies (e.g. Rate Monotonic, Earliest Deadline First, etc.) with different characteristics (preemptive or non-preemptive, static or dynamic, online or offline, optimal or heuristic, etc) the Scheduling Monitor is designed in a generic manner. One basic requirement on the Scheduling Monitor is that it can manage to monitor any scheduler implementation (of course any one that is compatible with the ORCOS specification) independent of the applied Scheduler. Hence, the parameters monitored by the Scheduling Monitor have to be generic to any ORCOS Scheduler to enable to determine the Scheduler's state:

- The Scheduler does not define any internal state. Additionally, it does not provide any generous scheduler-specific parameters that would allow to reason of being in an unstable or unhealthy state. However, if the Scheduler is executing, it consumes processor time and, thereby, produces processor utilization at the kernel side. So, a potential indicator for suspicious system behavior may be present in case the Scheduler is executing unexpectedly frequent. The execution of the Scheduler is based on timer interrupts. Timers are either calculated by the Scheduler itself to define time stamps for task instance's releases or are triggered externally by newly arriving tasks. In order to detect such unexpectedly frequent execution, the Scheduler Monitor measures the `Time Interval` between the Scheduler's execution and collects that values. The Scheduler Monitor stores, updates the values, calculates the average value for the `Time Interval` since system startup, but also considering the recent short-term history as well. Indications for unstable behavior may be obtained by a comparison between the current value for `Time Interval` and its average value. Any deviation may also be an indicator for a suspicious system situation. It is the task of system experts to define the correct conditions and set the right thresholds to identify such situations, as, in fact, this is a challenging task. Nevertheless, the Scheduler Monitor, as collecting that data, provides foundations for this.

- Besides monitoring the Scheduler as an OS component itself, the Scheduler holds knowledge about the tasks the operating system is executing. This knowledge can be exploit in the Scheduler Monitor to identify indicators for unstable states of tasks. Each task defines an arrival time, a worst-case computation time, and a relative deadline as static parameters. While executing, the Scheduler, and consequently the Scheduler Monitor as well, can gather the starting time, and related to this, the waiting time, the real current execution time, the finishing time and the respective response time.

  On the one hand, the Scheduler Monitor is intended to detect system failures such as a deadline miss or situations when the current execution time exceeds the predefined worst-case computation time (Obviously, these failures are based on faults in specification or schedulability analysis that should not happen if specifications and schedulability analysis are correct. But if these failures occur, the Scheduling Monitor will be able to detect them.)

  On the other hand, from these values collected, diverse conditions can be defined to identify suspicious behavior. An example for such a condition is: if the execution time of a task instance strongly deviates from the observed average computation time of that task, and coincidentally is close to worst-case computation time. This situation may not show up a failure, but if, for example, the worst-case execution time was never reached by other instances of that task, that particular deviation execution time may be an indicator for some suspicious behavior. An analogous argumentation can be formulated for monitoring the response time of task instances.

Last but not least, when monitoring the static parameters of tasks e.g. the relative deadline - which obviously should not change during the system's lifetime - the Scheduler Monitor is able to detect system manipulations and attacks in case of changes on these parameters.

The Scheduling Monitor collects parameters of tasks that are potential indicators to identify different types of suspicious or even malicious states of the tasks. The complete list of parameters collected by the Scheduler Monitor is provided in the Appendix C.1.

With the Scheduler Monitor, we are able to observe the performance of the Scheduler and the tasks. Furthermore, it allows to define conditions for safe behavior, suspicious or unreliable states and detect system failures such as deadline misses based on the parameters that the Scheduler Monitor collects.

Processor Utilization Monitor

Processor utilization is a critical factor in real-time operating systems. To guarantee meeting all system deadlines, the system requires a precise schedulability analysis and a thorough exploration of the processor utilization. The system's behavior must be fully predictable in terms of time and must not be overloaded. In [30], Buttazzo provides an exhaustive analysis of processor utilization (Equation 8.13, developed by Liu and Layland [70]) for different scheduling algorithms. However, Buttazzo performs the schedulability analysis based on the pure utilization originating from only the periodic tasks, of course based on worst-case assumption. The calculation of the processor utilization according to Equation 8.13 can only be applied for theoretical analysis, as from practical viewpoint, it completely ignores the overhead produced by the operating system.

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{8.13}$$

The schedulability analysis for system's that are dedicated to be put into operation must take into account the overhead produced by the operating system. The schedulability analysis performed at system design time delivers upper bounds for processor utilization. They do not reflect the real processor utilization as the system's runtime, but they can be exploit to deal as a critical indicator for determining the system's health state. In order to be able to evaluate the real processor utilization with respect to the statically determined thresholds, it is essential to monitor the processor utilization at runtime. This is implemented by our Processor Utilization Monitor.

The offline system analysis results in values for worst-case processor utilization for each system component (including the operating system itself and all tasks to be executed on the system). If, at runtime, the task's real processor utilization exceeds the offline calculated value, its behavior does not comply with its specification. In such a situation, we have an indicator for an unstable behavior. Hence, thresholds for processor utilization can be set based on the offline system analysis which deals as a reference value to verify the current processor utilization.

The general accumulated processor utilization of a system is defined by Equation 8.14 (t for the current time, *idle* reflecting the accumulated value for the length of time the processor was in idle state in t):

$$U = \frac{t - idle}{t} \tag{8.14}$$

or as the sum over all the execution times of the n components:

$$U = \frac{(\sum_{i=1}^{n} c_i) - idle}{t} \tag{8.15}$$

with c_i being the accumulated execution time of task or OS component i.

The task of the Processor Utilization Monitor is responsible to monitor the processor workload assigned to the kernel and to each task separately. It regularly checks and updates the collected data such as the current processor utilization of the current monitoring period, the short-term history processor utilization as well as the longterm processor utilization. Then, it matches the collected data against the system utilization thresholds or evaluates the specified conditions in order to generate the according signal (safe, warning, danger). The complete list of parameters monitored by the Processor Utilization Monitor can be found in Appendix C.2.

Monitoring the current processor workload provides benefit to another important aspect: based on the current processor utilization, the currently remaining processor utilization is known. This is very useful related to the system's ability to reconfigure itself. Assuming an unreliable system state is detected (caused by any system component) and a reconfiguration is required: then, the operating system should have enough resources available (which includes also processor time) to be able to reconfigure the system in order to restore the system into a stable or healthy state. On the one hand, it can be checked based on the current processor utilization whether there is enough processor utilization available to perform the reconfiguration. On the other hand, the fraction of the processor utilization required for reconfiguration should always be reserved for worst-case scenarios to guarantee system reconfiguration. The Processor Utilization Monitor monitors whether the worst-case processor utilization required for reconfiguration available, and raises a *warning signal* otherwise.

Memory Monitor

Memory is a restricted resource, especially, in embedded real-time systems. As the system has to behave fully deterministically, it is also valid for the memory usage. Monitoring the memory usage at run-time may prevent the system to run into an memory overload situation which, in real-time system, is unacceptable. Obviously, memory overload shall not happen if we could guarantee completely error-free specifications, programming, code verification and analysis as well as testing. Nevertheless, in practice errors are always possible and are possible to occur. Therefore, we apply our Memory Monitor to monitor the memory usage in order to detect suspicious situations in terms of memory management.

As introduced in detail in Sec 6.3.2, in ORCOS memory is separated into a kernel space, which is the memory space reserved for the operating system, and the user space that is the memory space available for the tasks. The kernel as well as each task get assigned their own Memory Manager.

The Memory Monitor regularly collects data of the overall memory usage, the kernel memory usage, as well as per each task. This collected data is matched against the available memory amount. If a task continuously allocates memory and never releases it, then the system or at least the task may run out of memory. But there are also other sources of dangers for the system. For example, if memory allocations are continuously executed (without any freeing of memory) but at the same time there is no increase in the memory usage (allocations of zero memory size), this might be either an indicator for a potential attack in forms of occupying the system (by empty allocation calls) or at least for a suspicious situation. Therefore, the Memory Monitor also collects data about the number of allocations as well as the number of calls of the free memory function in order to observe these calls, match them against the used memory size and, thereby, detect suspicious situations. The complete list of monitored parameters by the Memory Monitor can be found in Appendix C.3.

Communication Monitor

Communication is a fundamental part of any operating system. In RTOS, is must behave fully predictable in order to not endanger the deterministic behavior of the entire system. The Communication Monitor is responsible for the monitoring of the communication device of ORCOS and its performance. Therefore, it collects the values of the amount of data for `Upload` and `Download`, but also the according `Upload Speed`, `Download Speed` and the overall `Communication Speed` and checks whether and to what extent these values deviate from the calculated average values. Furthermore, it might be of importance if connection attempts fail. Therefore, the Communication Monitor implements a counter for the number of established connections and a further one for counting the connection attempts that failed. Indicators for a potential component failure may be found in various situations: the number of failed connection attempts in the recent history, if the number of failed connection attempts exceeds a threshold or when considering the rate of failed connections with respect to the number of established connections.

To be able to distinguish between a general problem in the communication device and a problem in a specific task, the Communication Monitor separately collects the communication-related data for the module itself as well as for each task. The parameters collected by the Communication Monitor are listed in detail in Appendix C.4.

File Manager Monitor

In ORCOS, any resource is managed in forms of files. The File Manager is responsible for organizing the access to files. For tasks that are supposed to access files or the file system, ORCOS offers system calls. The File Manager Monitor observes the performance of the File Manager in order to report its health state. Therefore, it collects the number of file accesses, but, moreover, it provides counters for each type of error related to file access (such as `cResourceNotOwned`, `cresourceNotWriteable` or `cResourceNotReadable`). An increasing number of an occurring error may be an indicator for an unreliable situation, especially if the error occured frequently in the recent history, and/or reaches a critical threshold. The File Manager Monitor collects this data set in a global manner as well as with respect to each task so that conclusions can be drawn considering the source of the suspicion. A complete overview of monitored parameters is given in Appendix C.5.

Device Driver Monitor

Devices in ORCOS are controlled by device drivers which provide an interface to offer access to the device's functionalities. Nevertheless, the ORCOS Module `Device Driver` is implemented as an abstract class in order to be inherited by specific device implementations.

Each device driver implementation is specific to the appropriate device. Therefore, no common health status parameters to be monitored could be identified.

However, the OS Health Monitor provides a module named Device Driver Monitor that is responsible for monitoring the health state of a device driver. Like its corresponding counterpart, the Device Driver Monitor offers an abstract framework that can be used to implement concrete Device Driver Monitor submodules for each device driver integrated in ORCOS.

Different means exist to obtain the health status of a hardware device:

- Some devices provide self-diagnosis features integrated into the chips to enable the detection of problems. If a device provides such a feature, then the device driver is able to obtain the data delivered by the self-diagnosis feature. Obviously, the corresponding Device Driver Monitor needs access to this data in order to collect it and get information about the device's health state.

- As the health parameters of a device are strongly depending on the specification and the purpose of the device, health parameters can only be identified by expert knowledge in the context of that device. Often, logical relations between attributes or conditions for parameters allow conclusions concerning the health state of a device.

  When implementing a Device Driver Monitor for a specific device driver, data relations and conditions have to be implemented individually. However, in order to generate and obtain health signals from these OS Health Monitor submodules, the Device Driver Monitor framework requires to access data in a generous manner. Therefore, the Device Driver Monitor framework defines methods for the access to the data and signals. The concrete implementations of data relations and conditions have to be encapsulated within these methods in order to unify the data extraction.

The abstract Device Driver Monitor framework provides a protocol for the implementation of different device driver monitors which leads to a reliable scalability of the OS Health Monitor. A unified interface allows to access the monitored data and the resulting health signal in a standardized manner. The methods defined by the interface are specified in the Appendix D.

The Component Monitor Modules internally collect all the data that contribute to the reflection of the operating system health state. Basically, the execution of a task has effects on the state of the kernel modules that in turn accounts for the operating system health state. Therefore, each of the Component Monitor Modules collects global component-related data that shows the internal state of the kernel module and task-related data that presents the parameters of that component with respect to the task's behavior. Based on these parameter values, each Component Monitor Module generates an individual local component-related health signal. Setting the boundaries between healthy, suspicious, or malicious system state requires a correct configuration of conditions and thresholds for the parameters in any of the Component Monitor Modules. In fact, it requires exhaustive analysis and evaluation with component-/ and system-specific expert knowledge. The local component-related health signals are combined to result in a global system health signal.

8.6.4 Signal Generator

The Signal Generator is the key part in the OS Health Monitor to generate the output signal that is offered for the classification in the anomaly detection module. It implements the functions and rules to analyze the data collected by the monitor modules and computes the current corresponding health signals. From the viewpoint of anomaly detection, it is the part in the system responsible for delivering the labels (see Chapter 2.2.1) that we call *health signals*. The rules that are specified in the Signal Generator are the core ingredients for the efficiency and reliability of the signals produced for our anomaly detection module. Component-specific expert knowledge is required that includes exhaustive system analysis to be able to correctly formulate these rules.

The signal provided by the OS Health Monitor was defined to take values in `green` (= safe signal), `yellow` (=warning signal) and `red` (=danger signal). For computing the signal, the Signal Generator operates on the data provided in the OS Health Monitor that was collected by the individual Component Monitor Modules. However, it is not straightforward or even impossible to define one rule that reflects the global system health state, taking into account the large number of parameters monitored (even though using only a restricted set of parameters). On the other hand, the diverse functionalities of the OS components lead to different parameters of different parameter types that reflect their purpose and specific behavior (the Scheduler Monitor collects scheduler-related parameters while the Communication Monitor collects communication-related data). For these different kinds of parameters, it is not possible to develop generic Signal Generator rules that could be generally applied to any Component Monitor Module. Each component has to be individually analyzed in detail based on its state parameters and results into a 'Component Monitor Module'-specific sub-signal (see Figure 8.24). The component-related sub-signals are accumulated and build up the global system's health signal. For the global health signal, 3 options are configurable at design time that enable the system designer more flexibility in implementing different strategies for reconfiguration:

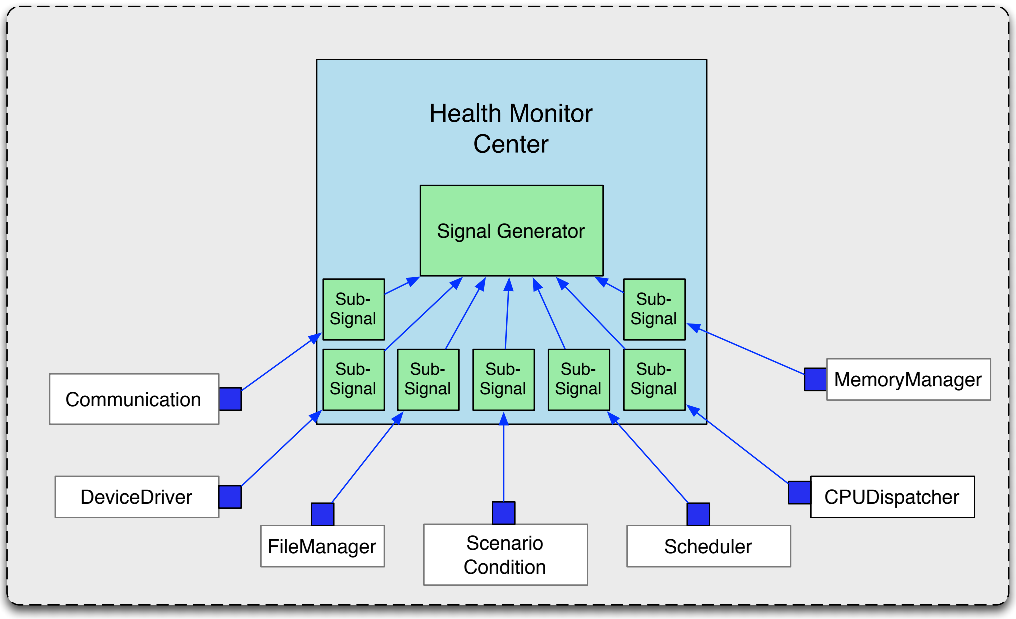


Figure 8.24: Signal Generator Design

1. The global system health signal is generated based on the maximum principle: if there is at least one sub-signal that is `red`, then the global health signal is set as `red`; if there is no `red` sub-signal but at least one sub-signal that is `yellow`, then the global health signal

is `yellow`; if there is no `red` and no `yellow` sub-signal, then the global system's health signal is set `green`.

2. The global system health signal is specified as a tuple counting the occurrences of the sub-signals: `health_signal` := (#red sub-signals, #yellow sub-signals, #green sub-signals).

3. The global system health signal is defined as a tuple of the component-related health signals as: `health_signal` := (health_signal_Scheduler, health_signal_ProcessorUtilization, health_signal_MemoryManager, health_signal_Communication, health_signal_FileManager, health_signal_DeviceDriver_i, health_signal_ScenarioCondition ...).

The first option provides one single value as a global indicator for the health state of the system.

The second option counts occurrences of the different health signal values delivered by the OS Health Monitor Components. Verifying the quantity of diverse signals enables the system to recognize whether a potential instability might have been propagated throughout the system components. The third option collects the sub-signals of the individual OS Health Monitor Components. Providing this kind of global health signal offers the possibility to localize the potential source of instability in the system based on the values of the component-related sub-signals.

Further potentials are provided by the OS Health Monitor components that collect generous component-related data (we call them global component parameters) as well as task-associated data. Based on this distinction, we are able to separately define rules that examine the state of a task and the state of the OS component and provide signals that are globally assigned to a particular OS component and signals assigned to the individual tasks. Let us assume that for one specific task the Memory Manager Monitor data exhibits indicators for an unstable memory usage of that task, but the global Memory Manager Monitor data is not in a critical state, then the OS Health Monitor's warning or danger signal is related to the task. Contrary, if the global Memory Manager Monitor's data is critical, which obviously will affect the single task-related Memory Manager Monitor data, then the Signal Generator will raise the according component-specific warning or danger signal.

This information can be exploited by the system's Controller responsible for the system reconfiguration: if a task is detected to behave unstable, then the Controller would decide to exchange that task while if the problem is globally present in an OS component, then the Controller should decide to reconfigure that system component.

The complexity of the system composed of a number of components leads to the fact that defining rules to obtain an accurate health signal is very challenging. Each component as well as its delivered parameters have to be examined thoroughly in order to be able to make a distinction between *safe*, *suspicious*, or *dangerous* state and consequently, raise the corresponding signal.

In the parameters of the Component Monitor Modules, the different types of (mainly numerical) attributes specified in 'Nature of Data' in Chapter 2.2.1 are present: interval and ratio attributes, predominantly available as time-ordered data. In fact, the data analyzing process to generate a *health signal* is related to anomaly detection in the data, detecting either *Point Anomalies* or *Contextual Anomalies*. Therefore, the rules we apply to the Signal Generator are mainly coming from the various anomaly detection techniques (introduced in Chapter 2.5) also implementing the different 'Design Decisions' (from Chapter 2.2.2).

For the data of the OS Health Monitor, we have characterized five different categories for data analysis:

1. **Static data analysis**

   There is a number of parameters that actually are static at system run-time, e.g. the `Relative Deadline` of a task. These static parameters are expected not to change unless in case of a system reconfiguration (e.g. exchange of a task by one with different parameters). Static parameters, however, are monitored by the OS Health Monitor in order to detect malicious system manipulations. The actual value is compared to the statically configured value stored in the OS Health Monitor Database. If a static parameter changes although no system reconfiguration was invoked, we can suspect a system manipulation that was not intended and, therefore, raise a *danger signal*. As long as there are no changes detected in the static parameters, the according signal remains *safe* (healthy state). For static parameter monitoring, *warning signal* is not supported.

   There is no history data required to be collected for static parameters so that the memory overhead caused by monitoring is minimal (one entry per static parameter).

   Samples of static parameters that are monitored by the OS Health Monitor (as implemented up to now) are:

   - **Scheduler Monitor:** `Relative Deadline` of each task (see Appendix C.1)
   - **Memory Manager Monitor:** `Memory Size` of each task and the kernel (see Appendix C.3)

   Static data analysis belongs to the class of 'Programmed Anomaly Detection' (see Chapter 2.2.2) that is dedicated to identify a *Point Anomaly*.

2. **Threshold-based analysis**

   Real-time systems require a system analysis at design time in order to be able to guarantee the schedulability, accurate timeliness and full dependability of the system. The analysis performed offline is based on worst-case assumptions on the performance (execution times, scheduling, and operating system overhead) of the system parts. From these worst-case assumptions, boundaries can be derived that specify thresholds for the parameter values at run-time. Like static data analysis, threshold-based analysis is also classified as a 'Programmed Anomaly Detection' technique (see Chapter 2.2.2) addressing the detection of *Point Anomalies*.

   As based on worst-case assumptions, exceeding a threshold shall, in general, lead to a *danger signal*. The definition of threshold-based *warning signal* is not supported by offline worst-case analysis and therefore, intuitively, not easy to be conducted. However, in the context of reconfigurability, the *warning signal* may become useful: Reconfiguration is a key feature in our operating system and the application environment that this anomaly detection approach is designed for. Reconfiguration is intended to be rarely performed (in case of failure or in terms of optimizations) as it requires a high amount of resources and temporarily may cause a decrease in system performance. The RTOS must guarantee full dependability in any case and, therefore, is obliged to reserve the according resources required for reconfiguration. Nevertheless, the reserved resources would stay unused for most of the time. But, by applying the ORCOS Flexible Resource Manager (see Section 8.1.1), ORCOS is allowed to employ resources that are dedicated to be reserved for worst-case scenarios. The FRM is responsible to allow such over-allocations only if the release of the resources can be guaranteed to be performed in such a manner that it does not violate any time or resource restrictions. Even if it is the FRM's obligation to manage such situations, over-allocation situations are critical for the overall system and require accurate performance. Hence, we have decided to raise a warning signal if resources reserved for system reconfiguration are borrowed to the tasks or the kernel.

Samples for thresholds set based on static analysis are:

- **Scheduler Monitor:**
  - The calculated worst-case execution time (WCET) is set as a threshold for `Current Execution Time`. If the current execution time exceeds this threshold, then the Scheduler Monitor sets the health signal to *red* (danger signal).
  - If the current `Response Time` exceeds the value of the `Relative Deadline`, a deadline miss happened and, therefore, the *danger signal* has to be raised. Therefore, we define the `Relative Deadline` as the threshold for the current `Response Time`.
  - We define a value called `Relative Nonactive Time` which is composed of `Relative Nonactive Time = Relative Deadline - Worst-Case Execution Time`. `Relative Nonactive Time` deals as a threshold for:
    1. `Waiting Time`; the current `Waiting Time` shall not exceed the `Relative Nonactive Time` (`Waiting Time` $\leq$ `Relative Nonactive Time = Relative Deadline - Worst-Case Execution Time`), as otherwise the Scheduler cannot guarantee that the task will be able to finish within its deadline
    2. `Preemption Time`; the current `Preemption Time` shall not exceed the `Relative Nonactive Time` (`Preemption Time` $\leq$ `Relative Nonactive Time = Relative Deadline - Worst-Case Execution Time`), as otherwise the Scheduler cannot guarantee that the task will be able to finish within its deadline
    3. `Blocking Time`; the current `Blocking Time` shall not exceed the `Relative Nonactive Time` (`Blocking Time` $\leq$ `Relative Nonactive Time = Relative Deadline - Worst-Case Execution Time`), as otherwise the Scheduler cannot guarantee that the task will be able to finish within its deadline
    4. sum of `Waiting Time`, `Preemption Time` and `Blocking Time`: the current `Waiting Time` $+$ `Preemption Time` $+$ `Blocking Time` shall not exceed the `Relative Nonactive Time` (`Waiting Time` $+$ `Preemption Time` $+$ `Blocking Time` $\leq$ `Relative Nonactive Time = Relative Deadline - Worst-Case Execution Time`), as otherwise the Scheduler cannot guarantee that the task will be able to finish within its deadline

  If the enumerated parameters exceed the threshold, an according health signal has to be raised. Its definition is left to be decided by the system designer with respect to whether the task is a hard (setting *danger signal*) or a soft real-time task (setting *warning signal*). Furthermore, for assigning the signal it is possible to take into account the (expected) `Average Execution Time`. If the `Average Execution Time` is much smaller than the WCET, raising a *warning signal* might be sufficient as the task will potentially be able to finish within the deadline (and *danger signal* otherwise).

- **Processor Utilization Monitor:**
  - For each task, the upper bound of the Processor Utilization is calculated offline by $\frac{C_i}{T_i}$ which is set as the threshold for a task. If the `Current Task Utilization` exceeds the threshold, then the task does not comply with the static analysis and a *danger signal* is raised.
  - For the overall system, the Processor Utilization has been determined by static system analysis as well. If the `Long-Term Processor Utilization` exceeds this value, a *danger signal* has to be raised.
  - For the reconfiguration, a fraction of the processor utilization U_{reconf} has to be reserved. If the current `Current Processor Utilization` exceeds the value of $1 - U_{reconf}$, then at least a *warning signal* is raised. The decision whether to raise a *warning signal* or a *danger signal* in that case is left to the system designer.
- **Memory Manager Monitor:**
  - The kernel and each task have their own memory space and a specified `Memory Size` which in fact acts as the threshold for the Memory Manager. If the current `Used Size` of a task or the kernel reaches the value of the `Memory Size`, a *danger signal* has to be raised in order to indicate a full memory usage.
  - For the reconfiguration, a dedicated amount of memory Mem_{reconf} has to be reserved. If the current kernel's `Used Memory` exceeds the value `Memory Size` $- Mem_{reconf}$, at least a *warning signal* is raised. The decision whether to raise a *warning signal* or a *danger signal* in that case is left to the system designer.
- **Communication Monitor:**
  - For each task that uses the communication device, a minimum communication speed is required to be able to guarantee its deadline. We set that required communication speed as the upper bound threshold. If the `Communication Speed` undercuts the required upper bound for communication speed, a deadline miss becomes probable and therefore a *danger signal* is raised.
  - In order to guarantee a deterministic behavior, in each period of a task an upper bound for `Communication Load` is determined. If the task's current `Communication Load` exceeds that value, a deadline miss becomes probable which leads to a raise of the health signal. The simplest solution is to immediately raise a *danger signal*. Another possibility is to allow the system designer to decide whether in that case to simply raise the *danger signal* or whether to match the `Communication Load` against the `Communication Speed` in order to decide to either raise a *danger signal* or only a *warning signal*. If the `Communication Load` exceeds the upper bound but the `Communication Speed` is higher than required, a *warning signal* might be sufficient.
- **File Manager Monitor:**

  The File Manager is responsible to manage the access to files. The File Manager Monitor mainly counts the number of errors occurring in the context of file accesses. Concerning file accesses, it is not possible to define thresholds based on static analysis. However, errors in file accesses will decrease the performance of tasks and are therefore essential. On the one hand, they may indicate a wrong implementation while on the other hand, they might indicate a potential attack that is dedicated to occupy the system (by senseless file accesses). Therefore, the system designer is allowed to set thresholds for acceptable occurrences for the various error types (see Appendix C.5) monitored by the File Manager Monitor. The thresholds can be set generally or even per period of a task's execution.

These example provide an excerpt of Signal Generator rules and show that setting thresholds for *danger signal*s based on static analysis is applicable. The challenge is how to define thresholds for the *warning signal*. This can only be decided by expert knowledge of the system designer who is aware of the application scenario and its environment.

3. **Analysis of evolving data**

   The values of the parameters are monitored regularly at dedicated time stamps. They are expected to differ at each monitoring point of time which, in most of the cases, is identical to one execution period of a task or a kernel module. The change of the parameter values may be of interest with respect to its tendency to evolve.

   Some examples for suspicious data progressions are:

   - **Scheduler Monitor**:

     The `Current Execution Time` is monitored by the Scheduler Monitor. A suspicious development can be identified if the execution time of a task tends to increase in each period of execution.

   - **Processor Utilization Monitor**:

     The current `Current Processor Utilization` is directly related to the execution time of a task. Hence, if the execution time of a task increases within each execution period, the processor utilization of the task will increase as well. Changes in `Current Processor Utilization` have equal significance as the `Current Execution Time`.

   - **Memory Manager Monitor**:

     The Memory Manager Monitor collects the `Alloc Size` of a task and the kernel per period. Let us assume the following situation: A task allocates a certain amount of memory in one period. Primarily, this is a valid action, especially in the task initialization phase. However, if the task continues to allocate memory in each period without releasing the same amount of allocated memory, it may lead to a 'run-out of memory' situation.

   - **Communication Monitor**:

     The `Communication Speed` is regularly collected. A continuous decrease of the speed may be suspicious as it may reach an upper bound at which the timeliness of the system may be disrupted.

   Of course, most of these situations - if they occur - are related to specification or programming errors that should have been detected by verification methods. But they could also arise because of system complexity coming from autonomous and dynamic system behavior.

   Detecting such changes in parameter values may be not critical for one or two successive values. But they may become critical if the values continue to change. Therefore, it is essential to identify such situations. To decide which signal to raise in these situations is not easy: Even if a value was increasing (or decreasing) in each period, it would be useful to know whether it will tend to increase in the same manner. It might continue to evolve in equal measure so that it will reach a critical value and cause dangerous circumstances. Contrary, its course may exhibit an asymptotic curve which is not targeted to yield into a danger situation. As the future progress of the value is unknown and not easy to predict, but may lead to a suspicious situations, at least a *warning signal* shall be assigned here.

In order to detect the course of change in parameter values, methods for time series analysis are powerful to be applied and provide mechanisms for clearly defining the appropriate health signal. Another approach is to use regression-based methods (see Chapter 2.5.4) that analyze the relationships of parameters in order to identify dependencies between the present parameters and, thereby, allow to make predictions about the development of values. It is left to the system's expert to decide which method to apply to which parameter set in order to ensure a reliable evaluation.

4. **Relationship analysis**

   There are health parameters collected by a Component Monitor Module that are interdependent. Any change in one parameter shall induce a corresponding change in another parameter.

   One example for such an interdependency is:

   - **Memory Manager Monitor**: The Memory Manager Monitor monitors the `Alloc Size` and counts each call of the system call `alloc` (or `new` as an alias function call, see Appendix A) by the parameter `Alloc Counter`. Analogously, it also counts the `free`- calls (or `delete`) and collects the information in the `Free Counter` and `Free Size`. Each call of `alloc` leads to a value for `Alloc Size` and an incrementation of the `Alloc Counter` (and for the free-related parameters in analogical manner). These parameters are interdependent as their change originates from the execution of one identical system call.

     Assuming that a tasks tends to call the `alloc`-function with an empty argument, it will lead to the `Alloc Size` $= 0$ received by the Memory Manager Monitor. (Of course, such a call is not expected. But it may occur and it might be caused by faulty programming or a potential attack on the system). Even if that behavior does not endanger the memory usage of the system, it provokes the system to be utilized by useless function calls and thereby occupies the system. It has to be detected in order to raise at least an *warning signal* to signify suspicious behavior.

     The resulting rule for this example is: If there is an increase in the `Alloc Counter`, the value of `Alloc Size` has to be > 0.

The Signal Generator provides rules to evaluate interdependent parameters in order to detect any suspicious behavior. Here, we have discussed an example that is intuitively reasonable. Further parameters exist that also show interdependencies. According rules for these parameters can be formulated.

However, in contrast to our example, it is possible that the interdependencies between the health parameters are not that clearly perceived . Their dependency may be unknown because the parameters may be not directly affected by the same system calls or even system components. To address this problem, the Signal Generator may use regression-based analysis in order to detect such interdependencies between health parameters and, thereby, enhance the accurateness of the health signals generated.

5. **Observation data analysis**

   For many parameters, their actual values at run-time cannot be predicted by offline analysis. They have to be measured online as they are affected by many different factors and interactions of the system parts in operation. Based on the monitored data, profiles can be derived that determine the *normal* behavior. Such *normal* profiles - often constructed by self-learning methods - could be expressed by a stable average value or a value range, by a distribution model, by patterns detected in the parameter value history, by clusters formed based on the observed data items, etc.

   Any observed parameter value that does not match the *normal* profile identifies a suspicious behavior that is declared as an anomaly which at least shall lead to generate a *warning signal*. In fact, the OS Health Monitor builds up its own internal anomaly detection mechanism for the observation data.

   To realize this, diverse anomaly detection techniques, introduced in Chapter 2.5, can be applied: Classification-based, Nearest Neighbor-based, Clustering-based, Statistical Anomaly Detection, etc. It strongly depends on the type of the parameters and its characteristics which anomaly detection method to choose. However, it is essential to take into account that the data amount the anomaly detection method will operate on is very restricted (as being applied in an RTOS for real-time embedded systems) which might reduce the pool of applicable anomaly detection methods.

   The decision which method to use is specific to each component's parameter and requires the expert knowledge. However, we provide some simple samples for 'observation data'-related anomaly detection by means of determining average values:

   - **Scheduler Monitor/Processor Utilization Monitor**:
     The Scheduler Monitor collects a number of parameters for which the average values can be determined and which are helpful to identify suspicious behavior:
     – Based on the monitored values of the `Current Execution Time` of a task, its `Average Execution Time` can be calculated. If the execution times of a task are similar and lay within a small range, then we can call the `Average Execution Time` as stable. If any further instance of that task exhibits a `Current Execution Time` that strongly exceeds the `Average Execution Time`, it might be an indicator that the system's or the task's performance has changed. The `Current Processor Utilization` is directly related to the execution time so that we can formulate an analogical rule for the task's processor utilization.
     Of course, if the `Average Execution Time` is detected to be close to the WCET, it becomes difficult to detect deviations that reflect suspicious behavior.
     – Based on the observations of the `Response Time` of a task, it is possible to verify whether the response times of the instances of that task are stable (lie within one interval). Then, the resulting average response time may deal as an indicator to evaluate the task's execution performance.
     Of course, the effectiveness of this rule is strongly related to the scheduling algorithm applied as the response times for single instances may also vary (e.g. when applying a fixed-priority scheduling policy like RM where its is possible to state that a task with statically-assigned highest priority will immediately execute as soon as it arrives. In contrast to that, applying a dynamic scheduling technique such as EDF, the priorities of the instances are different and lead to different schedules).

These are two examples how the average value might be significant to identify changes in task's performance. It is possible to formulate such average value-related rules for all the parameters of the Scheduler Monitor (`Waiting Time`, `Blocked Time`, `Preemption Time`, etc) if a stable behavior of a tasks can be taken for granted.

- **Communication Monitor**:

  The `Average Communication Speed` is calculated based on multiple monitored values of the `Communication Speed` and defines its *normal* value. If the current `Communication Speed` strongly deviates from the average value, a problem in the communication device may be the cause.

It is possible to calculate the average values for any kind of health parameters as one means to detect a task's or the component's behavior differing from its *normal* profile. Based on these average values (assuming that they are gained from stable and rather similar performance of the task's instances) outliers can be detected that are reflected by raising a *warning signal*. An alternative for our simple approach to use average values (in case they are not reliable), is to consider the median values, density-based anomaly detection, or statistical anomaly detection like histogram-based approaches (see Chapter 2.5.4) and match the current monitored data against that model.

However, the observation-based analysis does not only rely on statistical calculations. In particular, in case of strongly varying parameter values, statistics are not significant. Another approach to detect deviations in observation-based methods is to first identify behavioral patterns from the obaservation data. Pattern detection can be performed by using regression-based or sequence data analysis (Chapter 2.5.8). One example for pattern-based anomaly detection may be:

- **Memory Monitor**: The amount that a task allocates in a period is monitored in the `Alloc Size` and the `Memory Usage` of that task. Let us assume that the memory allocation in a task J_i follows a particular pattern such as: In an initial state, the task J_i allocates per period a dedicated amount of memory size. After reaching an internal threshold, the task may switch into a different state which changes its behavior concerning the memory allocations: for example, after n periods of allocation, the task could release all the allocated memory or start releasing step-wise some amount of allocated memory in the following m periods.

  From this behavior, a pattern for the `Alloc Size` and the `Memory Usage` values could be extracted to define its *normal* profile. The current behavior of the task in terms of memory usage shall be evaluated against this obtained *normal* profile.

Observation data-based analysis allows to define a *normal* profile for the behavior by applying different models. Determining the adequate model is the first challenge that the system designer has to face as a wrong choice will lead to an inaccurate model of the *normal* profile. The second challenge is to assign an appropriate anomaly detection technique based on the selected model as it must be efficient for small data sets and and work on restricted resources.

As the problem of implementing *Observation data*-based analysis is very complex, we are not able to provide a clear answer in form of specific Signal Generator rules here. This topic is left to the system and application designers that have knowledge about the specific application, the OS, and its application environment and are able to consider all contributing factors.

The different rule categories introduce different levels of accuracy: First, we can distinguish between *hard* and *soft rules*: *hard rules* are those that lead to a clear identification of a failure (and result in a danger signal). *Soft rules* are those that lead to an identification of a suspicious situation that cannot clearly be assigned to a dangerous state but potentially may lead to it. Rules defined in *Static data analysis* and *Threshold-based analysis* are exclusively *hard*. The remaining three categories allow to define both: *hard* as well as *soft* indicators. On the one hand, it depends on the characteristics of the concerned parameters whether it is possible to define rules that definitely point out a failure situation and such conditions that reveal a suspicion. On the other hand, it depends on the system designer/developer, his/her expertise and ability to define the rules or find appropriate algorithms and methods with qualities to classify the collected data into safe, suspicious and dangerous. For *hard rules*, we can assume a high quality of accuracy, while for *soft rules*, we are not able to make a general assumption.

The second implication on the accuracy is governed by priorities applied to the rules between the different categories. The *hard rules* detect dangerous system states. Hence, if any of the conditions defined by the *hard rules* is valid, a *danger signal* must be raised, without any ranking order between the rules. Hence, all *hard rules* are assigned a unique priority at highest level. It is different for *soft rules*: *Soft rules* can have different significance with respect to the accuracy of the applied method, they even may be competitive. To illustrate this, let us make use of the example of increasing memory usage per period. In the *Analysis of evolving data*, such a successive increase of memory usage may be defined at least to lead to a warning signal. When applying the *Observation data analysis*, this successive increase of memory usage may be detected as a *normal* behavioral pattern that shall be classified as *safe*. In order to get a clear health signal from the Signal Generator, it is essential to define an order of priority for the defined rules at least at the level of rule categories.

One possibility for such a prioritization of *soft rules* is to give *Observation data analysis* highest priority - as it relies on the learnt system behavior, then *Relationship analysis*, and at lowest priority *Analysis of evolving data*. Of course, this is only possible if the method applied in *Observation data analysis* is more efficient and reliable than the remaining ones.

With the *hard rules* identifying dangerous system state, the Signal Generator can be considered as a failure detector. But moreover, with its *soft rules*, the Signal Generator becomes an internal anomaly detector that is able to identify suspicious situations or states of the operating system components at an early stage.

We provided some samples or simple rules as examples, but of course, the given examples are incomplete. For anomaly detection, different models and anomaly detection approaches can be applied, including all kinds of anomaly detection principles: programmed as well as self-learning, supervised, semi-supervised as well as unsupervised. By this, *Point Anomalies* as well as *Contextual Anomalies* can be addressed. However, to formulate the Signal Generator rules correctly, and ensure a certain level of reliability and efficiency, the Signal Generator can only be realized by expert knowledge and appropriate methods for system analysis. This strongly effects the precision of the health signal that the OS Health Monitor delivers to the anomaly detection module. To give an example for integrating expert knowledge, in our case study (see Chapter 11), we have implemented a Signal Generator for an IR Sensor Monitor that is an implementation of the Device Driver monitor.

Based on the health parameters associated either globally with an OS component or specifically with a task, the Signal Generator is able to track down the source of the threat or instability. This offers great potentials to support an integrated system controller. The detection of the potential source of threat supports the controller's reconfiguration decisions that is considered as one means of a reaction on a threat.

8.6.5 Scenario Condition Framework

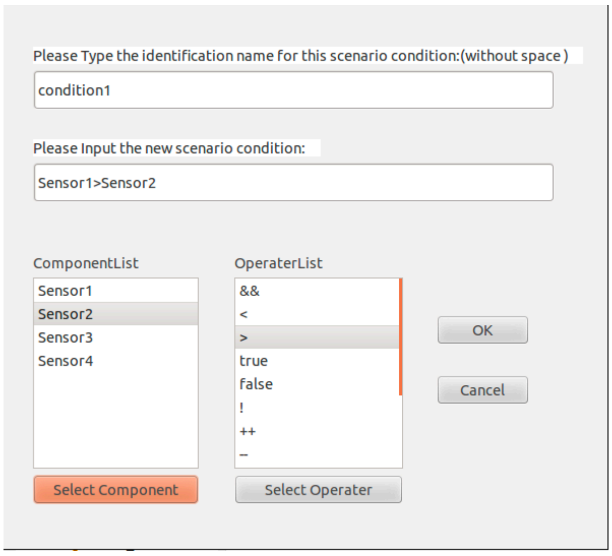
For each Component Monitor Module, we have identified parameters to be monitored in order to be able to evaluate the system's health state. In the Signal Generator, rules are implemented to produce the health signal based on the values of the monitored parameters. However, we already have emphasized that the choice of appropriate parameters and, moreover, the definition of rules for the Signal Generator strongly depends on a thorough analysis and is not possible to be conducted without expert knowledge. In our implementation of the Component Monitor Modules and the Signal Generator, we have worked in our fundamental system-specific knowledge without having involved detailed component-related expert knowledge. A fixed specification of parameters and rules for generating the health signal may introduce some deficiencies:

- From an expert's viewpoint, the list of parameters or the rules for signal generation that we have specified are not complete.
- The parameters or rules may not be suitable for a particular application scenario. Specific working environments may require specific parameters and conditions that may result in different health signal generator rules.
- The global health signal is derived from the local component-related health signals. These, in turn, are constructed based on the parameters of only that component. However, indications for dangerous situations may be discovered when combining parameters (or parameter conditions) from different monitor components.

To give experts and developers the opportunity to incorporate their knowledge and, thereby, enhance the OS Health Monitor, we have developed the Scenario Condition Framework. It allows to define further parameters, conditions and rules a posteriori without the need to directly include source code or modify the existing code of the ORCOS kernel or the OS Health Monitor. (Allowing users to work directly in the source code of ORCOS is risky and unacceptable with respect to the required system's dependability.) Therefore, the Scenario Condition Framework offers a user interface in forms of an editor that was integrated into the SCL Editor (ORCOS offline configuration tool, see Section 6.2).

In the Scenario Condition Editor (see screenshot in Figure 8.25), the user is able to select attributes that are available in ORCOS, define logical operations (such as $<$, $>$, $=$, $!=$, etc.) or thresholds for the attributes that have to be checked at operation time in order to guarantee a health state. Offering a form to the users ensures that the syntax of the formulated scenario conditions is correct. In order to integrate these conditions into the OS Health Monitor, some further phases are required:

After having defined the conditions, they are extracted by a Syntax Translator and translated into a specific format. The Syntax Translator is equipped with a Dependency Builder that verifies whether the user-defined conditions conform to the existing ORCOS dependencies. (As ORCOS is fully configurable, different implementations of kernel modules can be selected in the current configuration. They even can be completely activated or deactivated. Hence, it is important to check whether the attributes used in the conditions that can be of different OS components are available in the current ORCOS configuration.) As last step, the Syntax Translator transfers the conditions into an ORCOS-readable SCL Configuration file. This SCL Configuration file is bounded into the set of files for compilation. After the ORCOS compilation, the OS Health Monitor is retooled with the Scenario Condition Monitor consisting


Please Type the identification name for this scenario condition:(without space)

condition1

Please Input the new scenario condition:

Sensor1>Sensor2

Figure 8.25: Screenshot of Scenario Condition Editor ([69])

of a Condition Checker that includes all the user-defined conditions and a local Signal Generator. The described workflow of the Scenario Condition Framework is illustrated in Fig 8.26.

At run-time, the generated Scenario Condition Module is responsible for verifying the defined scenario conditions. However, the conditions can only result into either being true or false so that, in contrast to the other monitor modules, the Scenario Condition Monitor can only output two different signals:

- **safe signal** - if the scenario condition is true
- **danger signal** - if the scenario condition is false

The output signal of the Scenario Condition Monitor is treated as any other local health signal and contributes to the overall system-wide health signal.

With the Scenario Condition Framework, we offer potentials to flexibly extend the OS Health Monitor in a user-friendly and safe manner.

8.6.6 OS Health Monitor Database

The number of parameters that can be monitored is huge when consolidating all the possible parameters of the Component Monitor Modules listed. Furthermore, referring to a set of the rules for the Signal Generator (Section 8.6.4), considering only the current snapshot of parameters is not sufficient to make assumptions on the system's health state. There is a number of signal generating rules that take into account history data in order to evaluate the change in the values over a course of time. The data collected by the health monitor modules has to be centrally organized and stored in order to be analyzed for the purpose of signal generation. This is done in the OS Health Monitor Database that is responsible to hold data of the health parameters.

Storing that huge amount of parameters including their complete history (a snapshot of the parameter values is gathered in each execution period) is not possible in such resource restricted environments that we address with this anomaly detection approach. Memory usage must be bounded for any system part and, hence, for the OS Health Monitor as well.

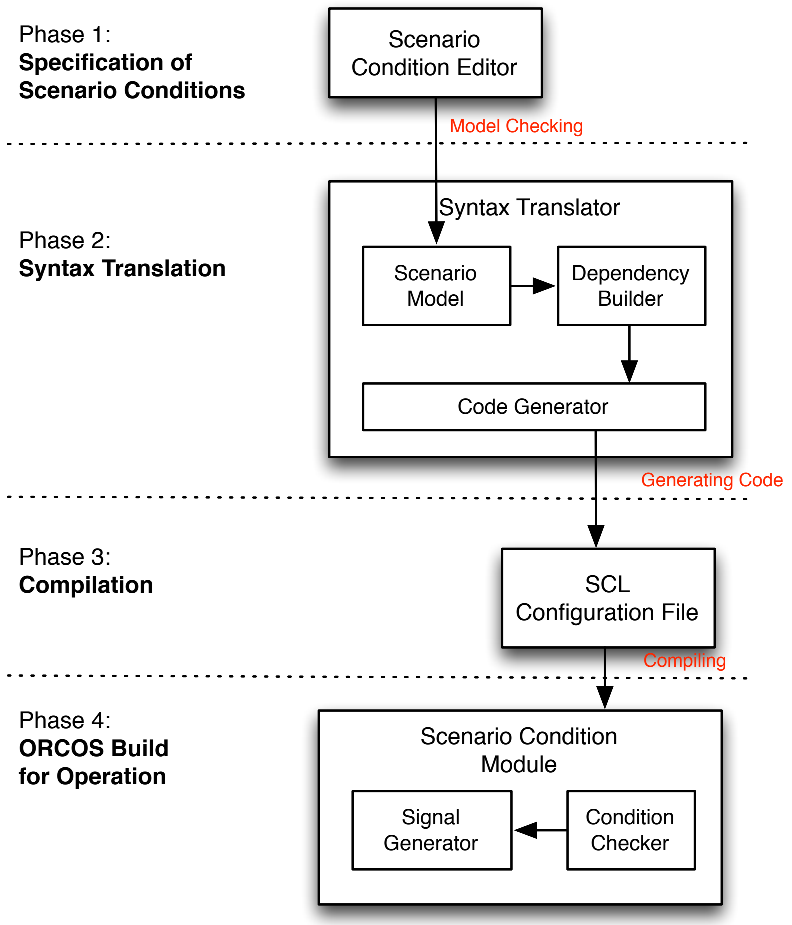


Figure 8.26: Scenario Condition Workflow ([69])

On the one hand, ORCOS offers the possibility for fine-grained configuration. At configuration time, the system developer has the opportunity to configure (add or remove) each parameter of the OS Health Monitor separately. Unused parameters (those parameters for which no signal generating rules have been specified) can be removed from the set of monitored parameters. With this option system developers can primarily reduce the memory requirement of the OS Health Monitor Database.

On the other hand, for determining the course of changes in the parameter values, only the recent history data is required. Usually, old history data that is not related to the current system state can be considered as outdated in the current context and, therefore, has no significance in terms of the evaluation of the current system health state. Consequently, the OS Health Monitor as a system component reflecting the current system state does not need to store the entire history of parameter values. As only a restricted number of values from recent (short-term) history are of interest, we decided to use fixed-size ring buffers for the health parameters that require to store history data. As a means to address the problem of resource restriction, this allows us to fully control the size of the OS Health Monitor Database and it prevents unbounded memory usage.

We call the applied data structure containing the set of configured health parameters of a Component Monitor Module with its associated history buffers the **health matrix** of a Component Monitor Module.

Even if the size of the health matrix can theoretically be determined, most of the monitor modules store task-related data. Hence, the real size of the health matrix is depending on the number of tasks that are executed by the operating system which - in case of ORCOS - is not finally set at compilation time because ORCOS allows dynamic loading of tasks during run-time. Consequently, static allocation of memory space for the OS Health Monitor Database becomes impossible as the required memory space is unknown. The OS Health Monitor has to dynamically allocate memory space for each task which, in fact, contradicts the second major requirement in real-time systems to ensure fully deterministic behavior. We have resolved this conflict by implementing a compromise: Based on a fixed memory usage of the health matrix for each task (that is defined by the used health parameters of the OS Health Monitor and the size of the ring buffers) we can allocate memory for the OS Health Monitor Database at the task's initialization phase. By doing so, we apply dynamic memory allocation which, of course, increases the task's initialization. However, as after the task's initialization all required memory is allocated, it will not effect the further execution times of the task after it becomes ready to be scheduled.

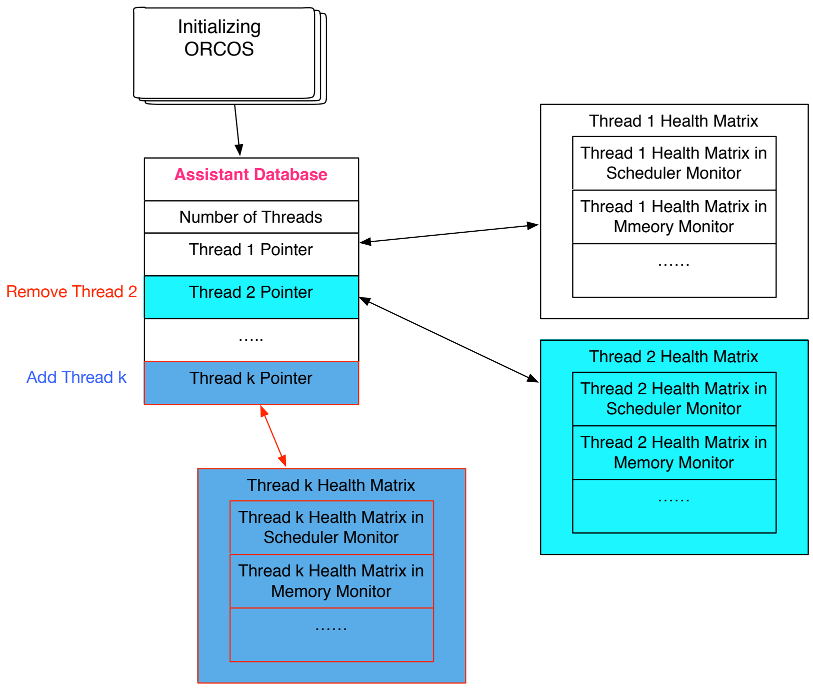

Initializing ORCOS
Assistant Database
Number of Threads
Thread 1 Pointer
Thread 2 Pointer
.....
Thread k Pointer
Remove Thread 2
Add Thread k
Thread 1 Health Matrix
Thread 1 Health Matrix in Scheduler Monitor
Thread 1 Health Matrix in Mmeory Monitor
......
Thread 2 Health Matrix
Thread 2 Health Matrix in Scheduler Monitor
Thread 2 Health Matrix in Memory Monitor
......
Thread k Health Matrix
Thread k Health Matrix in Scheduler Monitor
Thread k Health Matrix in Memory Monitor
......

Figure 8.27: Assistant Database (Source [69])

All information for the Signal Generator including all task-related health parameters are present in the health matrix which is completely stored in the OS Health Monitor Database. In order to manage the health parameters of a task and being able to extract the corresponding signals associated with a task in an efficient manner, we have designed an Assistant Database that holds the references to the task-associated health matrixes (of the activated monitor modules). The structure of the database is illustrated in Figure 8.27. In case the number of active tasks will change, the Assistant Database can be easily modified by adding, deleting or exchanging its entries. With the Assistant Database, we are able to provide access to the current task-related health parameters that are present in the running system.

8.6.7 OS Health Monitor Operation

The objective of the OS Health Monitor is to provide an output signal that reflects the system's health state. For this purpose, the OS Health Monitor has to undergo the following workflow

1. collect the system-related health data
2. store and
3. process the data.

The extraction and collection of the system-related health data is implemented by the Component Monitor Modules. The monitor modules gather the relevant data online at the point of time they are affected at the system's execution and store the data directly into the OS Health Monitor Database.

However, the OS Health Monitor Database as well as the data processing that is implemented by the Signal Generator must be controlled and managed. This is done by the OS Health Monitor Center.

OS Health Monitor Center

The OS Health Monitor Center is the core component of the OS Health Monitor framework responsible for managing its operation. The OS Health Monitor Center has assigned two main tasks:

1. management of the OS Health Monitor Database and
2. ensuring the delivery of the output signal by the Signal Generator.

The latter is strongly connected with the task of managing the Component Monitor Modules that have to provide their input to make it available to the Signal Generator.

The life cycle of the OS Health Monitor Center ordinarily consists of two phases: initialization phase and operation phase. In the initialization phase, the OS Health Monitor Database is initialized according to the health parameters selected in the configurations, the Component Monitor Modules are activated, the appropriate size of buffers as well as the current number of tasks are loaded into the system. Whenever a new task enters the system, the task-related database initialization is re-performed. Then, the activated Component Monitor Modules and the Signal Generator including its rules have to be initialized and prepared for their execution.

In the operation phase, the OS Health Monitor Center is responsible to control the usage of the database. This especially includes to ensure reliable access to the ring buffer when data is written to the OS Health Monitor Database but also the cleaning of expired data. In order to supply the Signal Generator with up-to-date data, the OS Health Center is responsible for triggering the Component Monitor Modules to preprocess their local signals and write their current data into the database (if not already happened online). The Scenario Condition Monitor is a monitor module, for example, (Device Drivers might be another example) that cannot consistently perform online processing as some conditions may be composed of parameters that are not synchronously changing and, hence, require the validation of the condition at a different point of time. By the triggering of the Component Monitor Modules, the OS Health Monitor Center enforces the Scenario Condition Monitor to update its data and its internal state. The data and the local output signal of the Scenario Condition Monitor are passed to the OS Health Monitor Database in the same manner as it is done for all the other Component Monitor Modules. After having triggered the Component Monitor Modules to update their data and their local signals, the OS Health Monitor Center drives the Signal Generator to execute computations. Figure 8.28 illustrates the components and actions of the OS Health Monitor Center.

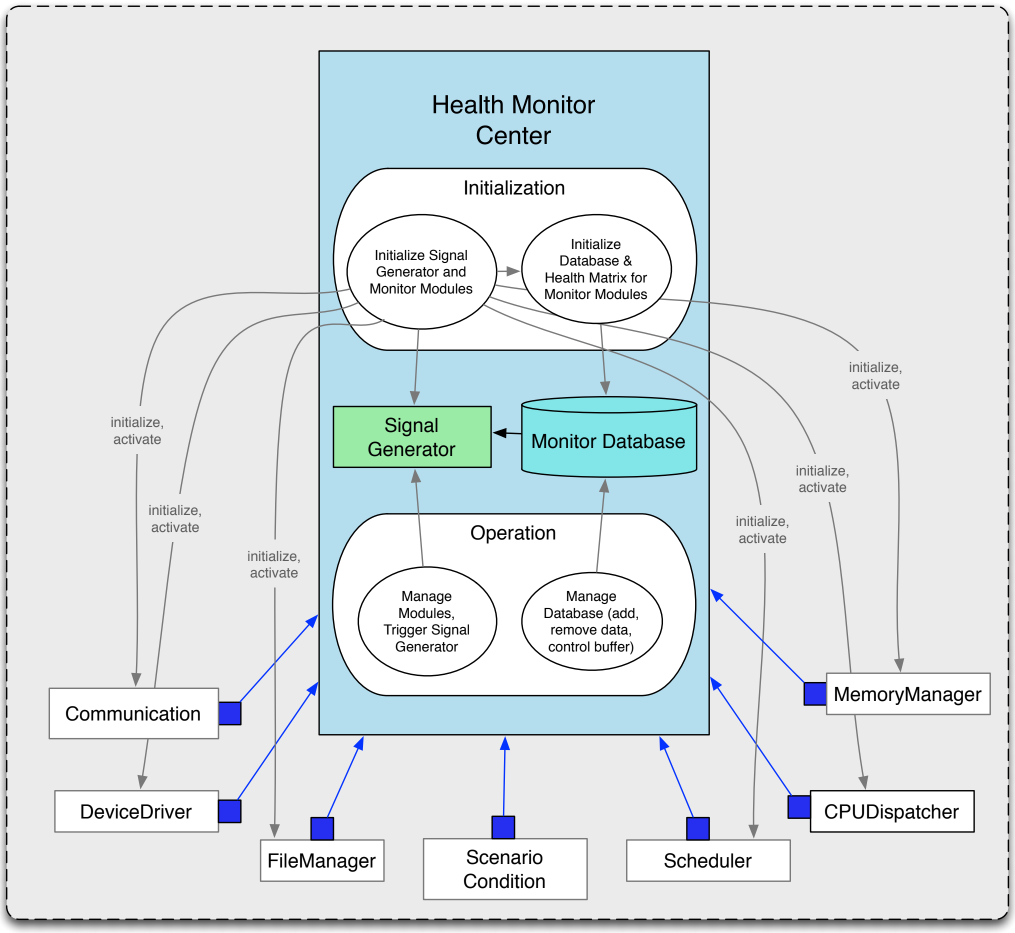

Health Monitor Center
Initialization
Initialize Signal Generator and Monitor Modules
Initialize Database & Health Matrix for Monitor Modules
Signal Generator
Monitor Database
Operation
Manage Modules, Trigger Signal Generator
Manage Database (add, remove data, control buffer)
initialize, activate
Communication
DeviceDriver
FileManager
Scenario Condition
Scheduler
CPUDispatcher
MemoryManager

Figure 8.28: Processing of OS Health Monitor Center

The OS Health Monitor Center is realized as an ORCOS workerthread (see Chapter ORCOS 6.3.1) in order to be regularly scheduled to guarantee fully deterministic OS behavior. The period of the OS Health Monitor Center has to be set according to the priority that will be given by the system designer to the OS Health Monitor.

8.6.8 Health Monitor API

The key intention of the OS Health Monitor is to supply the anomaly detection module with signals reflecting the system's health state. In order to get access to these signals (the global signal but also the local component-related signals), the OS Health Monitor provides an API.

The anomaly detection module is only allowed to read out data without being able to modify or manipulate the data. The data of the OS Health Monitor can only be modified by its internal components. Therefore, the API of the OS Health Monitor offers mainly `get`-functions and methods to trigger the OS Health Monitor to update the data and return the according health signals. The methods defined by the OS Health Monitor's API are listed in Appendix E.

8.7 Classification

The core approach of the presented anomaly detection method is the classification of observed behavioral patterns. Basically, the classification relies on evaluating the behavior in the context of the health signals delivered by the OS Health Monitor as introduced in Chapter 7. However, Chapter 7 merely introduces the classification idea, but leaves the question open what exactly is to be classified and how the context-related classification is conducted.

Generally, the classification is based on the concept of the traffic light principle as already applied for the health signals of the OS Health Monitor. The classification outcome takes on values from

- *green* for safe behavior,
- *yellow* for suspicious behavior, and
- *red* for detection of dangerous behavior.

8.7.1 Classification Entities

Behavioral patterns are the system call sequences executed by the running tasks and are the ones to be classified by means of the health signals. The execution of system calls has in turn effect on the values of the health parameters in different variations:

case 1 the execution of a system call has direct effect on a particular subset of health parameters

case 2 the consecutive execution of several system calls effects a particular subset of health parameters

case 3 a complete behavioral sequence (representing the execution of a task's instance in one period) has effect on the system's health parameters

With this differentiation into three cases, we have determined *three entities of classification* for our anomaly detection approach.

In reference to the example introduced in Section 8.5.4, we will illustrate the three cases of effects on the health parameters and the problems concerned with classification by using the sequences

Sequence S7 `7 7 0 1 8 $`

Sequence S8 `7 7 0 1 8 8 $`

as legal sequences associated with the code provided in Listing 8.1. Additionally, the following behavioral sequence is taken into account for the purpose of an illustration example (and may be considered as a potential outcome of a system reconfiguration):

Sequence S9 `7 ... 7 0 1 8 8 $`

System call ID 7 represents the `alloc`-system call while system call ID 8 stands for the system call `free`. The '...' (dots) in **Sequence S9** denote multiple occurrences of the system call 7. The remaining system calls are not relevant for the following discussion and are therefore not explicitly annotated.

Each single call of the system call 7 has effect on the allocation size of the task (see **case 1**) and, hence, on the health parameters of the corresponding OS Health Monitor Module. Each single call of the `alloc`-system call may lead to a memory allocation that exceeds a certain boundary so that a *warning* or even a *danger signal* might be raised by the OS Health Monitor.

case 2 refers to a consecutive subsequence that in correlation effects the health signals. **Sequence S9** illustrates such situation: even if one call of `alloc` will not endanger the system's health state, the series of calls, denoted by 7...7, might do.

Furthermore, considering **Sequence S7** - without the Ending Symbol $ -, it is a subsequence of **Sequence S8**. However, the effects of these two sequences on the system's health state are different: Without going to far into details and by ignoring the arguments of the system calls, we have to point out one main characteristic concerning the system calls `alloc` and `free`: due to their semantics, `alloc` and `free` construct a pair. From the point of view of memory usage, the call of `alloc` increases the memory usage, while a call of `free` decreases the memory usage. Usually, according to *good* programming standards and requirements, each memory allocated must be freed afterwards. Therefore, these two system calls are expected to occur in form of a pair in the behavioral pattern. The occurrence of these system calls as a pair in the behavioral sequence, assuming that `alloc` and `free` are called with the same arguments, neutralizes the memory usage of the behavioral sequence.

Referring to the presented example, **Sequence S7** contains two calls of `alloc` but only one call of `free`. Hence, the execution of **Sequence S7** will probably summarily increase the total memory usage of the task. In contrast, **Sequence S8** is **Sequence S7** extended by one call of system call 8, and, therefore, resulting into the same number of `alloc`- and `free`-calls which leads to a potential neutralization of the memory usage of that task. This shows that different behavioral patterns, even if one being a subsequence of the other such as **Sequence S7** and **Sequence S8**, have to be classified individually as described by **case 3**.

By the way, the latter example proofs the need for signifying the end of a pattern by introducing the sequence's ending symbol (indicated in Section 8.5.1) as it builds up the basis for the differentiation between behavioral patterns that are subsequences and other sequences in the knowledge base. The differentiation by means of the ending symbol allows behavioral patterns to be individually classified.

Representation of Classification Entities

The behavioral patterns are encoded in the Behavior Knowledge Base in form of Suffix Trees as described in Section 8.5. The structure of the Behavior Knowledge Base's Suffix Trees provides best prerequisites to differentiate between the three cases of classification entities. Each single system call is represented by a node in the tree (case **case 1**), the entire sequence is reflected by a path from root to an ending node, and consecutive subsequences are encoded within the path that can be identified being located in between branch nodes in the tree. (Referring to our implementation by using a trie, the occurring branch nodes are in fact the *real* nodes of the Suffix Tree. The branch nodes are the nodes that would persist as tree nodes, if the Behavior Knowledge Base would have been implemented according to its original specification of Suffix Trees, see Fig. 8.8, while the single system call nodes would be abolished by inserting their IDs into the edge labels.)

In order to ensure the differentiation of the defined three classification entities and their identification in the Behavior Knowledge Base's Suffix Trees, we have extended the structure of the implemented Suffix Tree by introducing a specification of different types of nodes: The basic class of a node is realized by `TreeNode`. Objects of this class usually represent the single system call node (see **case 1**). To distinguish *general tree nodes* from branch nodes and leaf nodes, specific types of classes have been defined for these nodes: `BranchNode` and `LeafNode` that inherit the class `TreeNode` as depicted in Fig. 8.29. Two successive branch nodes of a path enframe a pattern that occurs only in that consecutive manner and determines a minimal subsequence of the behavioral pattern. Hence, the posticous branch node identifies the end of the subsequence (as required in **case 2**). With reference to **case 3**, leaf nodes show up the end of a sequence and therefore identify the third entity of classification.

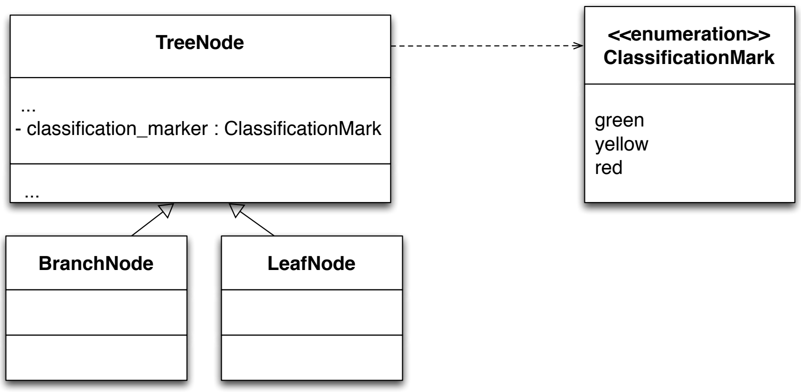


Figure 8.29: Class Diagram illustrating types of tree nodes

Classification Marker

All three entities of classification can be identified in the structure of the Behavior Knowledge Base. Based on this, the Behavior Knowledge Base offers best foundations to directly integrate the classification information. Each node in the Suffix Tree (independent of whether being an ordinary tree node, a branch node or a leaf node) is extended by one further attribute: the **Classification Marker** (see Fig. 8.29, classes equipped with the attribute `classification_marker`). The classification marker is assigned a value that corresponds to the configured global health signal representation explained in Section 8.6.4. That means, in correspondence to the global health signal, the classification marker is either

1. a single value taken from the enumeration set `ClassificationMark` = {green, yellow, red} that was obtained based on the maximum principle applied on the individual health signals generated by the contributing components (here: the Signal Generator rules), or

2. a tuple of values that represent the count of occurrences of the particular health signal values delivered by the contributing components `health_signal` := (#red sub-signals, #yellow sub-signals, #green sub-signals), or

3. a tuple representing the component-related health signals as: `health_signal` := (health_signal_Schedul

health_signal_ProcessorUtilization, health_signal_MemoryManager, health_signal_Communication, health_signal_FileManager, health_signal_DeviceDriver_i, health_signal_ScenarioCondition...)

The value of the classification marker is established on the basis of the behavior classification method implemented by the anomaly detection module and described in the remaining part of that chapter.

8.7.2 Classification Method

The objective of the classification method is to assign each classification entity instance a value for the classification marker. The basic entity is the tree node that represents one symbol of the behavioral pattern which actually is a representative for a single system call. Based on the fact that the execution of a system call directly effects on the values of a subset of health parameters, the classification of a system call - as a basic classification entity - is related to that subset health parameters. In order to perform the classification of applications' behavior in forms of the different classification entities, it is essential to first identify the effects of the system calls on the existing health parameters gathered by the OS Health Monitor:

Correlation of System Calls and OS Health Parameters

The impact of the system calls on values of the health parameters has been determined by code analysis of the system calls. The system calls offered by ORCOS are primary grouped according to the kernel functions they address, as depicted in Appendix A. In the following discussion on the effects of system call executions, this grouping is maintained as the analysis has shown that system calls grouped to one kernel function predominantly effect the corresponding health parameters of the Component Monitor Module of the OS Health Monitor that is associated with that kernel module. (The complete set of health parameters attached to the Component Monitor Module is listed in Appendix C.)

The following parts contain tables that illustrate the correlation of system calls and OS Health Parameters. In fact, a correlation means that a change of the health parameter value can be induced by the execution of the particular system call. In the provided tables, the correlation of system calls and OS Health Parameters is denoted by an 'x' in the table entry.

Health parameters that represent accumulations or average values, obviously will change because of changes in the base health parameters. However, these health parameters are not depicted here, as their changes are induced indirectly as a consequence of the change on the directly effected base health parameter values.

1. **Stream/File related system calls**

   System calls related to stream or file operations are addressed to the ORCOS resources and are mainly executed by services of the ORCOS File Manager. Therefore, the system calls belonging to this group directly effect the File Manager Monitor parameters as denoted in Table 8.1. Only system call `fcreate` effects the Memory Manager Monitor parameters as its execution is intended to create a new resource or file that has to be allocated in memory.

2. **Memory related system calls**

   System calls that are related to memory management imply `new` and `delete`. These two system calls affect only health parameters associated with the Memory Manager Monitor as illustrated in Table 8.2.

	File Manager Monitor							Memory Manager Monitor		
System Call	numberofResources	cOK	cError	cResourceNotOwnedTimes	cResourceNotWriteableTimes	cResourceNotReadableTimes	cInvalidResourceTimes	Alloc Counter	Alloc Size	Used Memory
fopen	x	x	x				x			
fclose	x	x	x	x						
fread		x	x	x		x				
fwrite		x	x	x	x					
fputc		x		x	x					
fgetc		x		x		x				
fcreate	x	x					x	x	x	x

Table 8.1: Mapping of System Call to OS Health Parameters: Stream/File related System Calls

	Memory Manager Monitor				
System Call	Alloc Counter	Alloc Size	Free Counter	Free Size	Used Memory
new/malloc	x	x			x
delete/free			x	x	x

Table 8.2: Mapping of System Call to OS Health Parameters: Memory related System Calls

3. **Task related system calls**

   The task related system calls are responsible for stoping, resuming or exchanging system tasks. The health parameters that are effected by these system calls belong to the Scheduler Monitor as shown in Table 8.3.

	Scheduler Monitor				
System Call	Start Time	Finishing Time	Current Execution Time	Preemption Counter	Preemption Time
task_stop			x	x	
task_resume					x
getSubtaskById					
changeRunningSubtask	x	x	x		

Table 8.3: Mapping of System Call to OS Health Parameters: Task related System Calls

4. **Thread related system calls**

   Similarly to task related system calls, thread related system calls effect parameters of the Scheduler Monitor. However, system call `thread_create` also effects health parameters of the Memory Manager Monitor. The effects of thread related system calls on health parameters are illustrated in Table 8.4.

	Scheduler Monitor								Memory Manager Monitor		
System Call	Arrival Time	Relative Deadline	Start Time	Current Execution Time	Preemption Counter	Preemption Time	Blocking Time	FinishingTime	Alloc Counter	Alloc Size	Used Memory
sleep				x	x	x	x				
thread_create	x	x							x	x	x
thread_run	x		x								
thread_self											
thread_yield	x		x		x	x					
thread_exit				x				x			

Table 8.4: Mapping of System Call to OS Health Parameters: Thread related System Calls

5. **Signal related system calls**

   The signal related system calls are addressing operations on exclusive resources that in ORCOS are administered by means of files. These operations are intended to block or resume the execution of threads with respect of the status of the required resource. Hence, these system calls, on the one hand, effect health parameters of the File Manager Monitor as well as the parameters of the Scheduler Monitor, on the other hand.

	File Manager Monitor	Scheduler Monitor			
System Call	numberofResources	Current Execution Time	Preemption Counter	Preemption Time	Blocking Time
signal_wait	x	x	x	x	
signal_signal	x			x	x

Table 8.5: Mapping of System Call to OS Health Parameters: Signal related System Calls

6. **Socket related system calls**

   Socket related system calls are a suitable example to show that a group of system calls does not only effect mainly one Component Module Monitor. In contrary, this group of system calls affects all the basic monitor modules including File Manager Monitor, Memory Manager Monitor, Scheduler Monitor and that Communication Monitor as illustrated in Table 8.6.

   In this context, system call `add_devaddr` is a specific system call: based on the fact that it enables to build up a socket connection to a local device by mapping its address on the socket, communication with that device (over the socket) may lead to effects on the health parameters of the specific device. However, these health parameters are not generally defined because they can only be specified in the context of that device. Even if there might be a correlation between system call `add_devaddr` and a device's health parameters, these health parameters are unspecified and therefore, they cannot be encountered here.

	File Manager Monitor	Memory Manager Monitor			Communication Monitor				Scheduler Monitor			
System Call	numberofResources	Alloc Counter	Alloc Size	Used Memory	cOk	cError	Upload	Download	Current Execution Time	Preemption Counter	Preemption Time	Blocking Time
socket	x	x	x	x								
connect					x	x			x	x	x	x
listen					x	x			x	x	x	x
bind					x	x						
sendto					x	x	x					
recvfrom					x	x		x	x	x	x	x
add_devaddr												

Table 8.6: Mapping of System Call to OS Health Parameters: Socket related System Calls

	Memory Manager Monitor						Communication Monitor				Scheduler Monitor		
System Call	Alloc Counter	Alloc Size	Free Counter	Free Size	Used Memory	Memory Size	cOk	cError	Upload	Download	Arrival Time	Relative Deadlin	Start Time
getTasktable	x	x			x		x	x	x	x			
create_task_physicalMemory	x	x	x	x	x	x					x	x	x
isDownloading			x	x	x		x	x		x			
preTaskloading	x	x			x		x	x	x				
createtask_physical_syscall	x	x	x	x	x	x					x	x	x

Table 8.7: Mapping of System Call to OS Health Parameters: System calls for Task loading

7. **System calls for Task loading**

   This group of system calls is designed for dynamic loading of new tasks at runtime. These system calls effect parameters of the Memory Manager Monitor, the Communication Monitor as well as the Scheduler Monitor as depicted in Table 8.7.

8. **Others**

   For the system calls belonging to the group of 'Others', no effects on the defined health parameters have been identified. In case of the system call `printToStdOut`, a device specific implementation is required. Therefore, this system call can potentially effect health parameters of the corresponding device driver (which are not specified yet).

Considering the types of OS Health Parameters, of course, *static health parameters* should not be effected by any system call execution as they are fixed over the entire system runtime.

Summarizing the presented correlation results, every base health parameter is directly effected by at least one system call and can be found in one of the mapping tables presented here. One exception are the `Time Interval` parameters that are set at the beginning of an execution period of the OS Health Monitor Center. System calls have no effect on the `Time Interval` parameters. Another exception are the parameters provided by the Processor Utilization Monitor. The base health parameters of the Processor Utilization Monitor such as `Idle Time` or `Kernel Occupying Time` are not directly initiated to be set by any system call. Moreover, their values can only be allocated whenever the ORCOS Scheduler is triggered.

As shown, only base health parameters are directly effected by the execution of system calls. However, the remaining health parameters, that are no base health parameters (and no static health parameters), are defined as compositions of base health parameters. Any change in the value of a base health parameter induces a change of all health parameters that are depending on that particular base health parameter. The dependent health parameters require a recalculation to update their values.

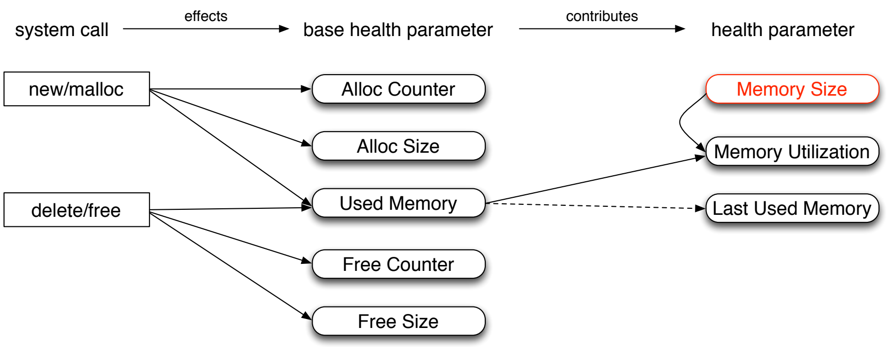


Figure 8.30: Effected OS Health Parameters by system call executions for Memory related system calls.

In fact, the health parameters that are not directly effected by the execution of system calls are affected in an indirect manner because of the chained effects of system call executions. The correlations are exemplarily depicted in Figure 8.30 for the memory related system calls. The graph shows the system calls on the left side and identifies their effects on the base health parameters. From the base health parameters, the effects on the remaining health parameters are illustrated. The red-marked health parameters are emphasizing static health parameters as those should never be modified by chained effects.

Such graphs for visualizing the correlations of system calls and health parameters can be constructed not only for memory related system calls but for any of the system call groups even though they are not provided in this work. In fact, in the anomaly detection module, the correlations are implemented in forms of mapping tables that contain all the interdependencies of health parameters and system calls.

By the chained effects, every health parameter is actually affected by some system calls (at least one) as, in the analysis, correlations with system calls (at least one) have been identified for every base health parameter (besides the exceptions explained above). We can determine the subset of health parameters that are effected by a particular system call (by following the paths of system call effects on health parameters illustrated in Figure 8.30). Vice versa, the subset of effecting system calls on a particular health parameter can be determined by the subset of system calls that effects the base health parameters that the particular health parameter depends on. We can determine this by following the reversal paths in Figure 8.30 starting from the health parameter to its effecting system calls.

Of course, the presented results represent a snapshot of the current ORCOS release. They reflect the implementation specifics of ORCOS' system calls and the specification of the parameters collected by the OS Health Monitor. Any change in the implementation of the system calls, any extension of the ORCOS' System Call Interface by novel system calls as well as any extension of the parameters monitored by the OS Health Monitor require a renewed analysis of potential changes and necessary extensions of the mappings presented here.

An example of such an extension is provided in the case study presented in Part 11. For the application discussed in the case study, we have defined application specific system calls and added a device specific Component Monitor Module to the OS Health Monitor. In this context, we have analyzed the newly-created effects of the introduced system calls on the health parameters (see Section 11.2.7).

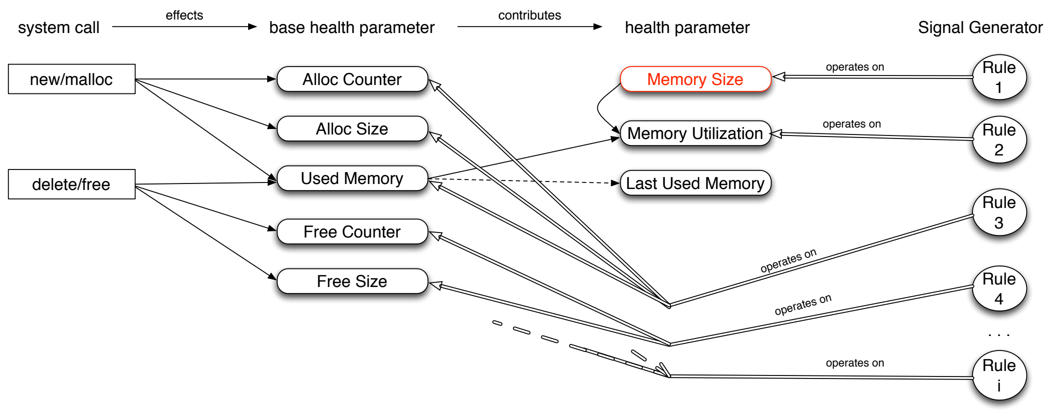
Identification of Signal Generator rules

Referring to the objective of our classification method, classification requires to equip each classification entity with a health signal. Still, we concentrate only on the basic classification entity: how to classify a system call. On the one hand, the anomaly detection module has knowledge about the health parameters effected by system call executions. On the other hand, the OS Health Monitor implements rules for generating health signals that operate on a subset of health parameters. The classification method implemented in the anomaly detection module combines this information as illustrated in Figure 8.31a. The extracted health parameters set effected by a particular system call serves as a basis to determine which of the rules defined by the Signal Generator are effected by the system calls. Actually, each rule that operates on at least one health parameter of the effected health parameter subset is a rule affected by the particular system call.

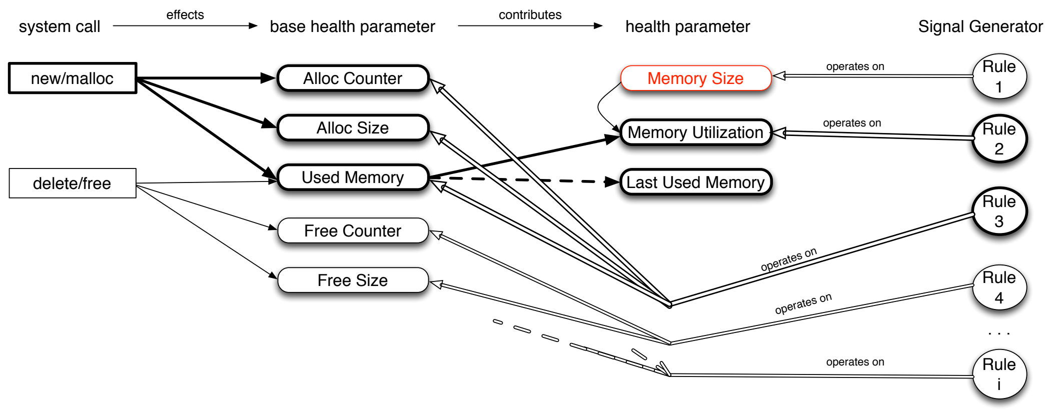
Hence, this procedure allows to assign a subset of Signal Generator rules to each system call as emphasized in Figure 8.31b for the system call `new/malloc`. The correlations of system calls and the effected subset of Signal Generator rules are integrated into the anomaly detection module by means of mapping tables.

Classification Procedure

The foundations for the classification of the behavior are the identified effected Signal Generator rules assigned to each system call. Processing the identified Signal Generator rules



(a) Correlation of rules and Memory related system calls



(b) Determining the subset of Signal Generator rules for system call `new/malloc`

Figure 8.31: Correlation between Signal Generator Rules and System Call Executions

- of course, according to the priorities and weighting as described in Section8.6.4 (Signal Generator) - delivers a health signal. This health signal builds the source for the Classification Marker that was introduced to the nodes of the Suffix Tree (see Section 8.7.1) as it is intended to hold the outcome of the health state evaluation.

In short, the basic procedure for classification is defined as following:

1. collect the effected health parameters and update their values
2. identify the effected Signal Generator rules
3. evaluate the health state by processing the identified Signal Generator rules and deliver the corresponding health signal
4. return the health signal to the Classification Marker of the node representing the classification entity

The obtained health signal is computed on a subset of Signal Generator rules. Therefore, the health signal, as obtained by evaluating the identified effected subset of rules, can only be considered as a *partial system health state*. Furthermore, in most cases the subset of Signal Generator rules is again only a subset of the rule set associated with the effected component(s), as emphasized in the example in Fig. 8.31b. Hence, the obtained health signal actually represents the current partial health states of those system components implied in the signal evaluation.

According to the procedure described above, this health signal provides the value to be set to the classification marker of the Suffix Tree nodes. These nodes, however, are representatives for different classification entities of a behavior sequence. In this context, the classification procedure requires a further specification of the different purposes of the classification entities and its associated classification markers:

1. **Tree Node**

   An ordinary tree node represents one symbol of the behavioral pattern which is a representative for a single system call executed. Its classification marker, consequently, is associated with the execution of that particular system call.

   To obtain a health signal for a particular system call, it is necessary to make use of the subset of Signal Generator rules assigned to that system call. According to the classification procedure described above, the effected health parameters have to be updated after the execution of the system call. Then, the assigned subset of Signal Generator rules is processed on the updated values of health parameters in order to evaluate the health state and deliver a health signal. The classification marker in the tree node of the system call preserves the health state value.

   This procedure, if immediately processed after the system call execution, allows to implement individual classification of every system call. The current partial health signal can be directly associated with the particular system call execution. Therefore, it leads to a classification of the behavior sequence in a symbol-by-symbol manner.

   In the context of classifying anomalies, the symbol-by-symbol evaluation of system calls enables to detect *Point Anomalies* (see Chapter 2.2.1).

2. **Branch Node**

   A sequence located between two branch nodes represents a unique and coherently executed subsequence of the behavioral pattern. A branch node declares the ending symbol of a coherent subsequence of the behavioral pattern. As already stated, consecutive system calls, on the one hand, may reinforce their effects on the health parameters and, consequently, may lead to a different health signal than each single system call On the other hand, they may compensate their effects on health parameters and, therefore, neutralize some one-symbol-related health signals and produce a health signal associated with the entire subsequence.

   The purpose of the classification marker of a branch node is to hold the health signal for the corresponding subsequence it represents. In contrast to the tree node's classification marker, it is not sufficient to identify the effected health parameters by the system call represented by the branch node. Moreover, primary, it is necessary to identify the complete set of health parameters effected by all the system calls that are symbols of the subsequence that is intended to be evaluated.

   For classifying subsequences, the identification of the set of health parameters is different. It relies on the basic principle that each system call is assigned a set of effected health parameters. But, as the subsequence consists of multiple system calls, we have to accumulate the sets of effected health parameters: First of all, the anomaly detection module holds a binary array called `subsequenceEffectedHealthParameters` for the subsequence evaluation (having the length equal to the overall number of health parameters, with the indexes being the IDs of the health parameters and the content being a binary marker indicating whether the health parameter is effected by the currently evaluating subsequence). Initially, all entries in the binary array are set to `false`. This initial state of the array is always required at the beginning of a subsequence which is identified by either a node descending directly from root node or a node directly descending from a branch node. After the execution of a system call, the effected health parameters of that system call are marked as `true` in the `subsequenceEffectedHealthParameters` array. This marking is performed for the consecutive system call executions until a system call is executed that, in the Suffix Tree, is either represented by a branch node or a leaf node (that stands for the sequence ending symbol). By reaching such a kind of node, the ending of the unique subsequence is reached. The established `subsequenceEffectedHealthParameters` array now marks all the health parameters effected by the current subsequence. Based on this set of health parameters, the associated Signal Generator rules are identified and processed. The resulting health signal delivers the value for the classification marker of the branch node.

The classification procedure corresponds to the basic procedure defined above besides the method to obtain the effected health parameters.

3. **Leaf Node**

   Leaf nodes mark the end of a behavior sequence and hold the Sequence Ending Symbol (represented by the `thread_exit` system call). While including a new (previously unknown) sequence, the leaf node marks the last symbol included into the Suffix Tree. The classification marker in the leaf node is dedicated to hold the classification outcome of the complete behavior sequence (or of the complete behavior sequence executed until then, in case of including a sequence). To obtain the health signal for the sequences, and accordingly the value of the classification marker, we provide two options:

   1. update all components' health signals
   2. update only effected components' health signals

   The first option requires the evaluation of all Component Monitor Modules and their implementing Signal Generator rules. The evaluation delivers the global health state of the system after the execution of a task instance.

   The second option relies on the procedure as applied for the branch nodes: identifying the effected health parameters of the system calls executed by the current task instance. To realize this, a second binary array called `sequenceEffectedHealthParameters` is introduced to collect the effected health parameters of the sequence starting from the root node up to the leaf node. After reaching a leaf node in the Suffix Tree, the Signal Generator rules effected by the health parameters determined in the array `sequenceEffectedHealthParameters` are evaluated.

   Independent of which option was chosen, health signals related to the Scheduler Monitor and the Processor Utilization Monitor are of interest here in order to obtain a health signal that represents the overall system's health state caused by the behavior sequence. These health subsignals are integrated into the leaf node's classification marker. However, health parameters of the Scheduler Monitor and the Processor Utilization Monitor can only be updated when the ORCOS Scheduler is triggered because the value allocation of the health parameters is directly integrated into the kernel module's code. Its realization is explained in the next section dealing with the entire run-time process of the anomaly detection.

For all three types of classification entities, the classification procedure relies on the identification of the effected health parameters. This, however, differs with respect to the classification entity: for the single system call, it is only the set of health parameters effected by that system call, while for a subsequence and the entire behavior sequences, it is the accumulation of effected health parameters by all system calls contained in the subsequences or the entire behavior sequence, respectively. The identified set of effected health parameters determines the the effected Signal Generator rules in order to obtain a health signal.

Coverage of Classification Outcome

The health signal obtained by executing the effected Signal Generator rules is the input source for the classification marker of the respective classification entity. However, because of applying Suffix Trees, each classification entity exists multiple times as it is contained in the paths identifying the suffixes of the behavior sequence (of course, depending on its position within the behavior sequence). For example, Fig. 8.32 shows a Suffix Tree for the behavior sequence $S_j = 7\ 0\ 1\ 8\ \$$. For the classification entity `TreeNode` $S_j(2) = 0$, we have two nodes as representatives. For the `TreeNode`, the classification entity containing the sequences ending symbol \$ occurs four times, one node with the sequences ending symbol \$ generated for each suffix of the behavior sequence. According to the specification of the `TreeNode`, each node is equipped with the classification marker attribute, independent of being an ordinary `TreeNode`, a `BranchNode` or a `LeafNode`. Hence, it is the question whether each representative of a classification entity shall capture the classification outcome in its classification marker. Referring to our example, the question is concerned with whether for classification entity `TreeNode` $S_j(2) = 0$ that exists twice in the Suffix Tree both representative nodes have to be assigned a classification outcome produced by executing the classification procedure. This question becomes more relevant in the context of the `LeafNode` classification entity: Shall the classification outcome produced by the classification procedure classifying the entire sequence be assigned to each representative of the sequence's Sequence Ending Symbol (meaning all the four nodes in the Suffix Tree in Fig. 8.32)?

We will discuss this question by referring to the example using the sequence $S_j = 7\ 0\ 1\ 8\ \$$: Let us assume that $S_i = 0\ 1\ 8\ \$$ is a behavior sequence executed before ($i < j$). Furthermore, let us assume these sequences have different classification outcomes such as the sequence $S_i = 0\ 1\ 8\ \$$ is resulting in a `red/danger` signal while the sequence $S_j = 7\ 0\ 1\ 8\ \$$ is returning a `green/healthy` signal. These classification outcomes have been obtained in the context of the executions of the corresponding entire behavior sequences. The same is true for the classification entity `TreeNode` as $S_i(1) = 0$ may lead to a different classification outcome than the execution of 0 in the S_j as the second symbol of the behavior sequence. In order to preserve the context of the classification outcome of the classification entity, it should be ensured that classification outcomes are assigned to the classification markers of the nodes belonging to the path that determines the complete sequence. As a means to ensure this, we have introduced a further distinction of paths in the Suffix Tree: let us call the path identifying the complete behavior sequences from root to the leaf node the *main path*, while all paths and/or nodes belonging to the suffixes are called *reproductions*, see Figure 8.32. The *main path* is traced during

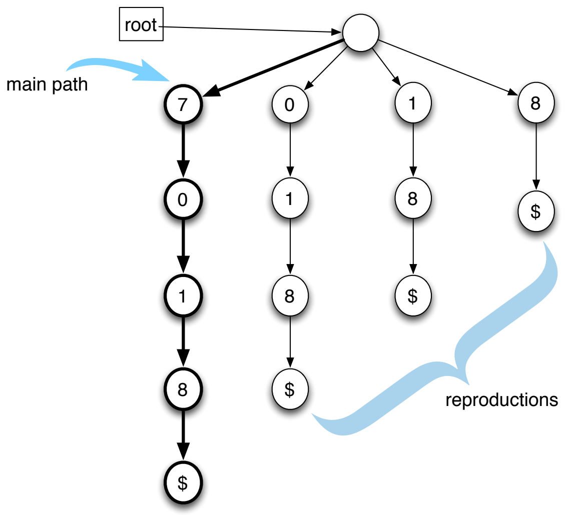


Figure 8.32: Illustration of the *main path* and the *reproduction* nodes on the example of sequence 7 0 1 8 $.

the application's execution. At the activation of a new task instance, the *main path* is reset to root. For each system call executing and extending the Suffix Tree, the *main path* pointer is set to the node in the path representing the complete behavior sequence. When executing the classification procedure after each system call, the classification outcome is therefore assigned only to the node of the current *main path* pointer. Thereby, the Suffix Tree is able to maintain (previously obtained) classification results of sequences that share the same paths (parts of or even complete suffixes) with *reproductions* of the current sequence.

8.7.3 Further Classification-affecting Factors

Up to now, the classification method in the Anomaly Detection Module is implemented as described above. The health signal and the classification marker obtained accordingly set in the Suffix Tree are evaluated on the basis of values of health parameters and their associated Signal Generator rules.

However, instead of purely adopting the health signal value to the classification marker, further factors exist that also may have impact on the value of the classification marker - yet not implemented, but already prepared by the implementation. Here, we offer a short discussion on some suggestions for enhancing the procedure to evaluate the classification marker value. A detailed discussion of open research issues in this context is provided in Appendix F.1.

1. **Previous Classification Marker Value** The previous value of a classification outcome related to a classification entity may be of interest for examining the health state of the current instance. Information about the previous classification marker value can be exploited to enhance the evaluation, reinforce values, and improve the precision of the obtained health signal. By incorporating the previous classification marker into the evaluation of the current classification marker, the classification outcome can become more sensitive and may assess an anticipated system's health state at an early stage. We introduce a further attribute to the `TreeNode` named `previous_classification_marker`. Whenever a node in the SuffixTree is reached by the current behavior sequence, first, the value of

the classification marker is transferred to the node's `previous_classification_marker` before the evaluation of the health signal related to the current execution is performed.

2. **Occurrence Counter** Information whether a behavior sequence is of high or low occurrence may be of importance in terms of classification. The number of occurrences of a classification entity can have an impact on weighting parameters in the Signal Generator rules as well as strengthen the weights of the rules themselves. Therefore, we have introduced an occurrence counter `visit_counter` into the `LeafNodes` in order to record its number of occurrence. Whenever the leaf node is reached by a behavior sequence, its `visit_counter` is incremented. By this leaf node's attribute, the number of occurrence of each predecessing node can be derived based on the characteristics of the Suffix Tree.

In this section, we present ideas and approaches on how information, on the one hand, available in the Behavior Knowledge Base and, on the other hand, extensible with low effort may contribute to the classification method. The decision how to integrate the previous classification marker, the occurrence counter and potential further classification-affecting factors (not presented here) into the evaluation of a behavior sequence and to choose an appropriate strategy for their impact is left to system designers and analysts because they have more specific knowledge with respect to potential correlations and threats, including an acceptable rate of false alarms, detection sensitivity etc.

In the case study, we have implemented some simple integration of the previous classification marker value into the calculation of the classification marker value concerned with the IR Sensor Device Driver Monitor (see Chapter 11).

8.8 Runtime Process of Anomaly Detection

The following sections discuss the coordination of the behavior of the anomaly detection in operation:

8.8.1 Anomaly Detection in Operation

The components that compose the Anomaly Detection Framework build up a logical workflow process derived from their responsibilities and the dependencies (of data output/input) between the components:

1. System Call Monitor to record the behavior sequences,
2. Behavior Knowledge Base in form of Suffix Trees to conserve the behavior patterns,
3. OS Health Monitor to provide health state signals for classification,
4. Classification Method as a part of the Anomaly Detection Module
   - 4.1 to perform the context-related (health signal-related) classification of the behavior entities and
   - 4.2 to store the classification results in the Behavior Knowledge Base, and
5. the Anomaly Detection Module to coordinate the actions of the Anomaly Detection Framework.

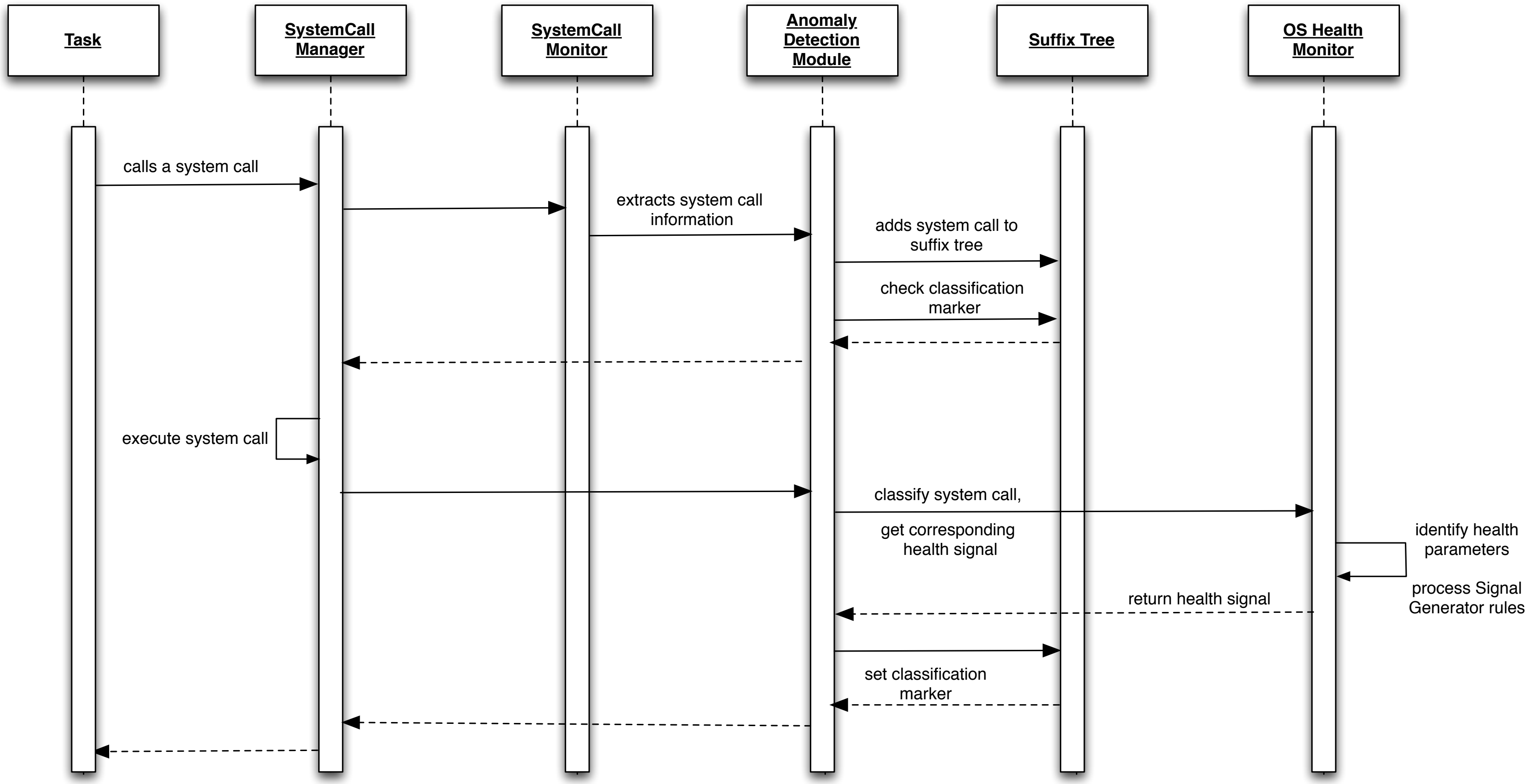

Task
SystemCall Manager
SystemCall Monitor
Anomaly Detection Module
Suffix Tree
OS Health Monitor
calls a system call
extracts system call information
adds system call to suffix tree
check classification marker
execute system call
classify system call,
get corresponding health signal
identify health parameters
process Signal Generator rules
return health signal
set classification marker

Figure 8.33: Classification Run Time Process.

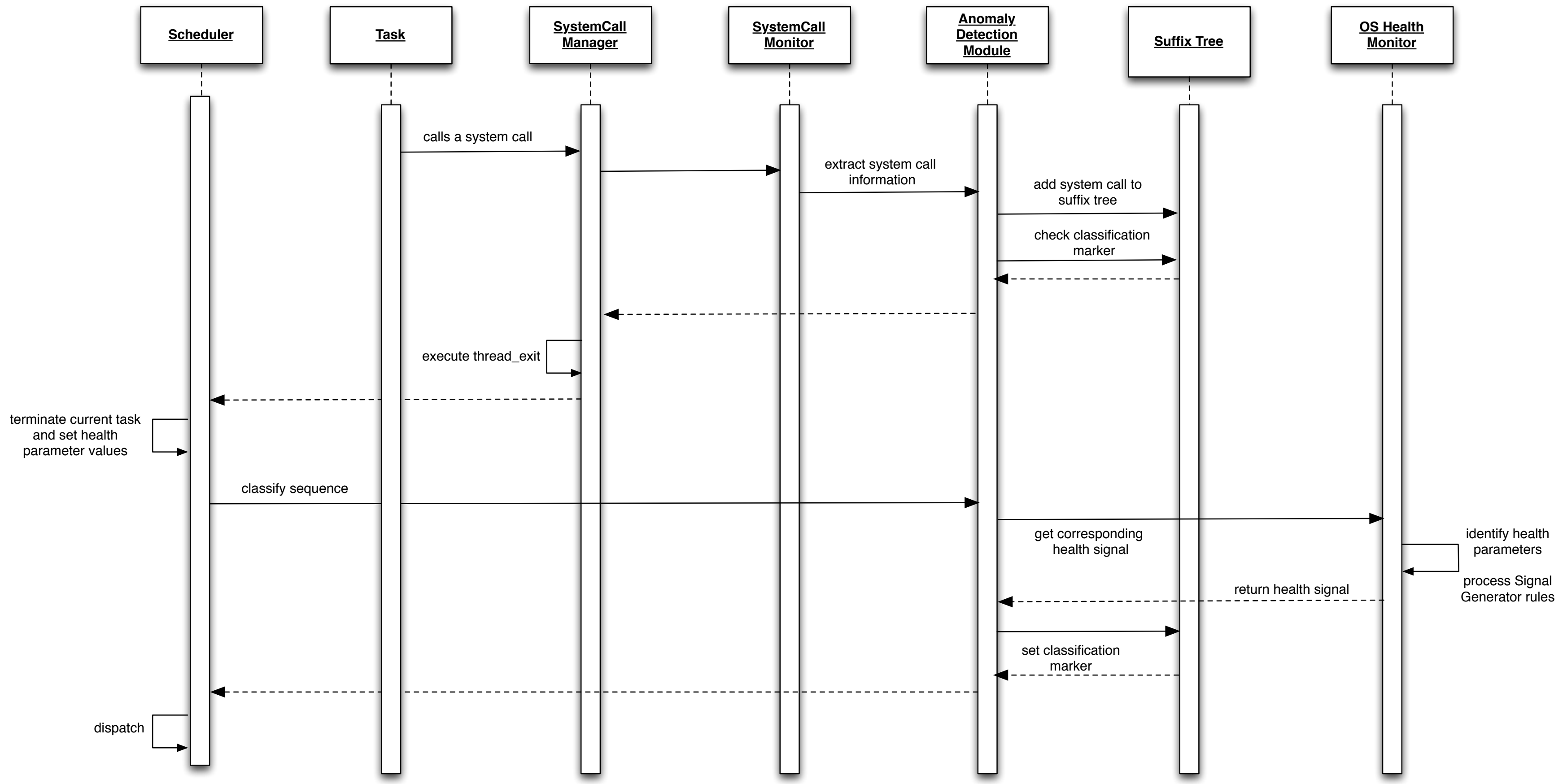

Scheduler
Task
SystemCall Manager
SystemCall Monitor
Anomaly Detection Module
Suffix Tree
OS Health Monitor
calls a system call
extract system call information
add system call to suffix tree
check classification marker
execute thread_exit
terminate current task and set health parameter values
classify sequence
get corresponding health signal
identify health parameters
process Signal Generator rules
return health signal
set classification marker
dispatch

Figure 8.34: Classification Run Time Process for Sequence Ending Symbol.

This workflow has been integrated into the real-time operating system ORCOS as illustrated in the Sequence Diagram in Fig. 8.33. This Sequence Diagram shows the workflow flow of processing a system call that is not the Sequence Ending Symbol $ (the `thread_exit` system call, respectively): Whenever a task executes a system call, the kernel's System Call Manager is triggered in order to serve this system call. Before the actual execution of the system call, the System Call Monitor extracts all information required by the mode that is configured. (In the case of our anomaly detection approach, it is only the system call ID that is recorded.) The data extracted by the System Call Monitor is forwarded to the Anomaly Detection Module. This in turn initiates the adding of the data into the Behavior Knowledge Base at the proper location in the Suffix Tree. If extending the Suffix Tree by adding a node is performed, the value of the classification marker is initially set to *green/healthy*.The node associated with that system call in the path determining the entire sequence is marked in the Behavior Knowledge Base's Suffix Tree as the *main path*. The value of the classification marker of that node is checked (if already existed it is associated with its previous classification) first, before the control flow continues and returns back to the System Call Manager that now triggers the execution of the system call. (The reason for this previous checking of the classification marker will become more clear in the following part of this section.) Generally, whenever a node is reached that was already existing, the value of the classification marker is assigned to the previous classification marker before the actions of the anomaly detection workflow are continued. This is done in order to preserve the previous classification outcome and to prevent losing this information by overwriting this value by the currently following classification. After the execution of the system call, the System Call Manager does not yet return back to the calling task. Moreover, the System Call Manager initiates the Anomaly Detection Module to classify the currently executed system call by means of the classification method. For getting a health signal associated with the current system call, the Anomaly Detection Module calls the OS Health Monitor to identify and update the effected health parameters and process the involved Signal Generator rules. The OS Health Monitor returns the health signal to the Anomaly Detection Module. This health signal is called by the Anomaly Detection Module to be set as the classification marker of the system call's corresponding node marked as *main path* in the Suffix Tree.With this, the anomaly detection process is finalized so that the Anomaly Detection Module returns back to the System Call Manager, that in turn returns back to the calling task.

There is one condition that will abort this control flow: This is at the step of checking the classification marker value after having located the system call in the Suffix Tree. As a means of ensuring dependability of the system, one objective is to prevent the execution of behavior sequences that lead to an unstable or even dangerous state. Whether the current behavior sequence tends towards such an unstable or dangerous state by executing a system call can be evaluated by checking the value of the classification marker associated with a former execution. If the node has already existed, then its classification marker has a value that is associated with the classification of a former execution of that behavior pattern and thereby, provides information on the health state resulting from its previous execution. If the previous classification marker of that node associated with the currently processing system call is *red/danger*, such a potential dangerous state is identified actually before the execution of the active system call. Consequently, in order to prevent a change-over to a dangerous system state, the execution of the system call is prohibited by the Anomaly Detection Module which thereby aborts the previously referred optimal workflow.

In case the system call is the Sequence Ending Symbol defined by `thread_exit`, the process of classifying differs slightly. This is illustrated by the Sequence Diagram in Fig. 8.34: Up to the system call execution by the System Call Manager, the anomaly detection process of `thread_exit` remains the same. However, as dedicated to terminate a task instance, `thread_exit` is not returning back to the calling task but is implemented to return back to the Scheduler in order to perform a context switch and to induce a scheduling decision. When the Scheduler is terminating a task, it writes the health parameter values related to the Scheduler Monitor Module and the Processor Utilization Module. These health parameters are required to evaluate the health signal related to the complete behavior sequence of a task instance. Therefore, before the Scheduler calls the Dispatcher to dispatch another task, it calls the classification method of the Anomaly Detection Module to initiate the classification of the Sequence Ending Symbol that corresponds to the classification of the behavior sequence. The classification is performed according to the process described above: updating the health parameters, identifying the set of effected health parameters, processing the effected Signal Generator rules and returning the obtained health signal to the *main path* pointer (here the leaf node containing the Sequence Ending Symbol) of the Suffix Tree.

Runtime Complexity

The Sequence Diagram (Fig. 8.33) illustrates the optimal flow when the anomaly detection is completely performed in an online manner. It introduces runtime overhead into the execution of each system call s_i. From the perspective of runtime considerations, this optimal flow constructs the worst case as it is maximally extending the runtime overhead. The resulting runtime overhead is composed of:

1. **Time to extract the system call information of s_i:** The time required for extracting the system call information by the System Call Manager (applying the basic mode for only extracting the system call ID and time stamp, see Section 8.4.1) is constant for any system call and will be denoted by $t_{SysCallMan}$.

2. **Time to extend the Behavior Knowledge Base by s_i:** This step is related to

   - at least: time to locate system call information in the Suffix Tree. This time is constant as it requires to check whether the current *main path* node has a child node with the newly arriving system call ID of s_i. The time for checking will be denoted by $t_{checkST}$.
   - and in worst case: time to add system call information to the Suffix Tree. As discussed in Section 8.5.6, the time for adding a system call s_i is governed by the position i of the system call within the sequence ($O(i)$) and will be denoted by $t_{addST_{s_i}}(i)$.

   The time for setting the *main path* pointer to the representative node of s_i is also constant and included into $t_{checkST}$. The resulting runtime overhead produced by extending the Behavior Knowledge Base is therefore $t_{checkST} + t_{addST_{s_i}}(i)$.

3. **Time to check classification marker value and to transfer it into the `previous_classification_marker`:** Checking the previous classification marker of the current *main path* note and setting its value to `previous_classification_marker` is constant for any system call and is denoted by t_{class_marker}.

4. **Time to classify the classification entity:** composed of

4.1 Time to identify effected health parameters: The correlation between system calls and health parameters is stored in arrays. Therefore, the identification of the effected health parameters requires constant time denoted by $t_{identifyHP\_syscall}$. For the classification entities `BranchNode` and `LeafNode`, additionally, this includes to update the according arrays `subsequenceEffectedHealthParameters` or `sequenceEffectedHealthParameters` (see Section 8.7.2) which also requires constant time denoted by $t_{updateHP\_arrays}$. Because it is not possible to predict the node type of the current system call s_i, we have to assume the worst case for the time overhead composed of $t_{identifyHP\_syscall} + t_{updateHP\_arrays}$ being defined as $t_{identifyHP} = t_{identifyHP\_syscall} + t_{updateHP\_arrays}$.

4.2 Time to update effected health parameters: After having identified the effected health parameters, the OS Health Monitor initiates the corresponding Component Monitor Modules to update the health parameters. The time required for performing the update depends on the number of health parameters and the execution times of the corresponding Component Monitor Modules. In worst case, it is the entire set of health parameters involving the execution of all Component Monitor Modules. We denote this time by $t_{updateHP}$.

4.3 Time to identify and process effected Signal Generator rules: The time required for identifying and processing the effected Signal Generator rules is depending on the set of effected Signal Generator rules. According to the argumentation above, in worst case, it is related to the execution of the entire set of Signal Generator rules. This time is denoted by $t_{processSG}$.

In fact, the steps of updating the all health parameters (step 4.3) and processing all Signal Generator rules (step 4.3) are successively performed by the OS Health Monitor Center when being executed as a schedulable Workerthread. Therefore, the sum of $t_{updateHP}$ and $t_{processSG}$ is equal to the runtime of the OS Health Monitor Center denoted by $t_{HealthMonCenter} = t_{updateHP} + t_{processSG}$.

4.4 Time to compute a health signal: The Signal Generator rules deliver sub-signals that are combined to a health signal for the corresponding system call s_i. The time required to compute the health signal is constant and will be denoted by $t_{healthsignal}$.

5. **Time to write classification marker value:** Returning the obtained health signal requires constant time and is included into the computation of the health signal in $t_{healthsignal}$.

The entire runtime overhead for a system call s_i is defined as a sum of the runtime overhead of the individual steps performed. Hence, we can define:

$$t_{AnomalyDetection}(s_i) = t_{SysCallMan} + t_{checkST} + t_{addST_{s_i}}(i) + t_{class\_marker} + t_{HealthMonCenter} + t_{healthsignal}$$

Apart from $t_{addST_{s_i}}(i)$, all ingredients of the runtime overhead are constant, so that the resulting time for processing a system call is $O(i)$.

The execution time of a task is extended by the runtime costs attached to anomaly detection for each system call that the task executes. Assessing the runtime overhead for a complete task, we have to consider the worst case execution of a task, which - from the viewpoint of the anomaly detection- is its longest possible behavior sequence $S_{n_{max}}$ with n_{max} system calls. Calculating the runtime overhead for a task is done by summing up runtime overhead of the individual system calls:

$$t_{AnomalyDetection}(S_{n_{max}}) = \sum_{i=1}^{n_{max}} t_{AnomalyDetection}(s_i)$$

This results in quadratic runtime $O(S_{n_{max}}) = O(n_{max}^2)$. By this, we can state that the runtime cost of the Online Anomaly Detection are bounded based on the number of system calls a task is executing. Performing the anomaly detection online allows immediate behavior classification and identification of suspicions, threats and dangers as soon as they occur. Therefore, the resulting runtime overhead becomes acceptable and justifiable. Of course, as an RTOS must guarantee to be fully deterministic, this runtime overhead has to be integrated into the Schedulability Analysis of the operating system, but also of each single task to be executed on ORCOS. The individual costs of each of these phases have been evaluated separately and are presented in Chapter 9.

Alternative Processing

The Online Anomaly Detection workflow introduces the worst case extension of runtime for a task as estimated above. There might be situations where this extended runtime could lead to unschedulability of the task set. To overcome this, a deviation from that optimal workflow is specified in order to reduce the runtime costs of a system call execution. This is realized by encapsulating the Anomaly Detection Module within a periodically activated Workerthread and is offered as an option that can be chosen in the system's SCL Configuration. In that case, the System Call Monitor extracts the system call information and collects them within its internal buffer. If the Anomaly Detection Module is selected by the Scheduler to execute, then the Behavior Knowledge Base request the collected system call information from the System Call Monitor's buffer, and processes the specified steps of inserting the behavior sequence into the Suffix Tree and performing the classification method. Choosing this option reduces the runtime overhead of the system call execution to the time related to system call monitoring $t_{SysCallMan}$. The Anomaly Detection Module as being executed in form of a Workerthread contributes by its utilization $U_{AnomalyDetection}$ (defined by its execution time $C_{AnomalyDetection}$ over its specified period $T_{AnomalyDetection}$) to the overall system utilization.

If applying anomaly detection in this way, it is important for the system designer - who chooses this option - to be aware of the fact that the classification precision is degraded: The classification method relies on the health parameters. These, however, can not be associated with a particular system call anymore, but moreover define the health state of the task at the particular point of time when the Anomaly Detection Module is executed. Furthermore, the health parameters generated during the application task execution can be outdated or even overwritten by successively executing system calls so that the context of the classification may get lost. Because the classification is performed according to the specified period, it is only possible to calculate the health signal for the task's current execution state. This state is determined by the task's latest system call being monitored that is represented by the *main path* pointer in the Behavior Knowledge Base. Only this node is assigned a classification marker. This leads to many leaks in terms of classification markers in the Behavior Knowledge Base. Additionally, in between the executions of the Anomaly Detection Module, potential suspicious or malicious health states may occur in the system that will be missed by the classification or even masked because of successively proceeding execution of the application task.

A further discussion on alternating the anomaly detection process for reducing the time overhead set on a system call is provided in Appendix F. Here, some ideas are presented that require more research and can become topics of future research.

8.8.2 Reconfiguration

Causes for reconfiguration are either requirements on the performance (optimization, dynamically adapted requirements), or the exchange of a defective system part. A reconfiguration is signalized by the *inflammation signal* as described in Chapter 7.2. It is induced by the Controller (see Section 8.2.3). We distinguish between two different kinds of reconfiguration:

1. reconfiguration of the operating system
2. reconfiguration of application tasks

Even though both kinds of reconfiguration change the system behavior, they have different effects on the classification. When reconfiguring the operating system, the application behavior will maintain. A change in the behavior of the operating system components will have an impact on the system state and therefore, on the health parameters. Contrary, a reconfiguration of the application tasks induces a change in the task behavior and can generate different behavior patterns. That leads to an expected change of the structure of the Suffix Tree. By a reconfiguration of an application task, the health state of the operating system is initially not effected, but can mutate because of the change in the application behavior. Additionally, a reconfiguration of both, the application tasks and the operating system, can be performed at the same time, if required by the decision strategy of the Controller.

In order to differentiate between these two kinds of reconfiguration, the *inflammation signal* is specified to identify the source of reconfiguration: The *inflammation signal* is defined as a tuple of

$$Infl = (OS, APP).$$

with OS: the flag for reconfiguration of the operating system and APP: the signal for reconfiguration of the application task.

OS is a binary variable set to 0 in case of no reconfiguration and set to 1 if a reconfiguration has taken place.

APP is in turn a tuple with n binary elements, one for each registered task in ORCOS sorted according the task IDs. A reconfiguration of the task $ID = 3$, for example, is signalized by $APP = (0, 0, 1, 0, ...)$. As the Controller is able to reconfigure multiple tasks at once, APP can contain multiple elements with a flag of 1. The *inflammation signal* $Infl$ is set by the Controller at the point of time a reconfiguration is induced.

Depending on the kind of reconfiguration determined by the respective value of the *inflammation signal* $Infl$, different alternations in the classification strategy are initiated:

1. Reconfiguration of the operating system, $OS = 1$:

   A reconfiguration of the operating system (or a single component) leads to a change in the system performance and its resulting health state. Previous health parameter values become invalid, especially, component-related health parameters and in particular, history and average values. Because a reconfiguration acts like a (at least partial) system restart, the reconfiguration-related health parameters are set back to their initial value by the OS Health Monitor.

Consequently, all classification results obtained on the basis of these health parameters will also become invalid, meaning the related classification markers in the Behavior Knowledge Base. For example, nodes that have been classified as *red/danger* in the previous system configuration, may receive different classification results as one of the reasons for the reconfiguration could have been these *red/danger* signals. Assuming that a task is executing a behavior pattern that formerly lead to a *red/danger* state (*red/danger* marked node in the Suffix Tree). Based on the classification workflow, however, the classification marker of each node is checked before the system call is able to execute. A *red/danger* classification marker, as defined in Section 8.8.1, would lead to an abortion of the task's execution. However, after the reconfiguration, the resulting health state is expected to be different and, therefore, instead of aborting its execution of the system call, it has to be evaluated from scratch. In order to interfere this procedure and to enforce the classification procedure to continue after a operating system reconfiguration, it is essential to invalidate the (old) classification marker values in the Suffix Tree.

The Anomaly Detection Module is responsible to invalidate the classification markers in the Behavior Knowledge Base. After receiving the *inflammation signal* from the Controller, the Anomaly Detection Module initiates to reset all classification marker values in all Suffix Trees.

The resetting procedure therefore traverses all nodes in all paths of the Suffix Tree and sets the classification marker to *green/healthy*. The traversing procedure is implemented by means of an ordinary tree-traversing algorithm *Depth-first search* (DFS)([41], Chapter 3) to reach each node. The complexity of this procedure is bounded to the complexity and structure of the Suffix Tree. The resulting runtime costs, of course, are mainly determined by the number of nodes, which is an individual characteristic and specific to the task's characteristics. The (worst-case) runtime costs caused by the resetting procedure have to be taken into account into the costs of reconfiguration.

In this approach, the classification marker values of all system calls gain a reset. It is not considered that only one OS component has been reconfigured which may involve that all the remaining ones will maintain their health states. By resetting all classification marker values, the information of health states of the component not affected by the reconfiguration will get lost.

An alternative approach to address this problem is to only reset those classification marker values of the system calls that are affected by system reconfiguration. This approach is prepared in the framework, but not completely implemented yet. To identify the system calls which classification markers are to be reset after the system reconfiguration, the correlation graph assigning health parameters to system calls (introduced in Section 8.7.2) has to be employed. First, it is necessary to identify the health parameters that are related to the reconfigured component. All system calls that posses a correlation to the identified health parameters are declared to be effected by the system reconfiguration. This information can be obtained by backwards following the correlation graphs.

However, as not implemented yet, this approach has not been evaluated: An open question remains whether a reconfiguration of one component may produce positive by-effects to other components such that suspicious or even dangerous states situations will be eliminated? In such a case, the set of system calls that require a reset of their classification marker will have to be determined by an other method.

2. Reconfiguration of an application task, $APP = (..., 1, ...)$:

A reconfiguration of a task is performed by the Controller by means of switching between the task's profiles (see Section 8.1.1). The profiles implement different strategies of the task in terms of its objective. They realize different algorithms for solving the same problem as well as different requirements in resource usage. Every profile defines its individual behavior. However, the degree of deviation in behavior between the profiles is specific to the task's implementation and specific to the defined profiles. After a reconfiguration, the task's behavior may totally change, but it may also be very similar - if not equal - to the behavior of the task's former configuration. There is no general assumption on the change of the behavior after a reconfiguration. Statements on profiles' behaviors can only be made by the application's designers.

From the viewpoint of the Behavior Knowledge Base, a total change in behavior will make the formerly constructed Suffix Tree obsolete. All patterns established by the former configuration will no longer occur while all new patters must in any case be built into the Suffix Tree. Contrary, in case the behavior remains equal or similar, the Suffix Tree established before the reconfiguration can prove beneficial as behavior sequences produced after the reconfiguration will already be present in the Suffix Tree.

After a task's reconfiguration, the Anomaly Detection Module is informed by the *APP* value of the *Infl* signal, which task is effected. The Anomaly Detection Framework provides different options for handling the Suffix Tree after a task's reconfiguration.

- use the present task's Suffix Tree
- build up a new Suffix Tree from scratch

The option can be configured by the application developer in the SCL configuration. Although having the knowledge on the specific task, choosing the adequate option for a task is a challenge: On the one hand, it is not predictable which of the profiles the Controller will choose when deciding for reconfiguration as there are numerous dynamic factors that contribute to its decision. On the other hand, tasks may implement self-adaptivity so that even the behavior of each single profile may become unpredictable.

Another aspect is how to deal with already determined suspicious or dangerous paths in the Suffix Tree. As the reconfiguration is applied to the task and primarily does not effect the operating system, the state of the system is maintained and, hence, health parameter values (despite of those that are task-related) are still valid and will remain. Classification entities (system calls or complete behavior sequences) that lead to a suspicious or even dangerous system state before the reconfiguration, can potentially do similarly after the reconfiguration. Hence, it can be of value to maintain these information for supporting the behavior evaluation after the reconfiguration.

If the option to use the same Suffix Tree is chosen, then all information are preserved anyway. In case of building up a new Suffix Tree, the Anomaly Detection Framework offers a further option:

- keeping all paths with node whose classification markers are *yellow/warning* or *red/danger*

This is implemented differently from purely generating a new Suffix Tree after the task's reconfiguration. Initially, the Suffix Tree of the task is maintained. All paths in the Suffix Tree are identified that completely consist only of nodes with a classification marker value *green/healthy*. The nodes of these identified paths are deleted from the Suffix Tree so that only such paths remain that contain information on suspicious or dangerous behaviors. We call this remaining Suffix Tree the *Suspicion Suffix Tree*.

A last challenge is to decide how to incorporate the knowledge of suspicious or dangerous behavior patterns into the behavior evaluation performed after the reconfiguration. Behavior sequences that have been classified as *yellow/warning* or *red/danger* in the former task's configuration (with a high probability) potentially can but do not need to lead to the same classification outcome. For example, other argument values in the system calls, that are not taken into account and, therefore, are not examined by the anomaly detection approach presented here, can probably lead to a different system health state than in the former executions.

So, the question remains whether to allow tasks to continue their execution if executing a behavior pattern that was previously classified as dangerous? For this purpose, the Anomaly Detection Framework offers two options to handle this question:

- pessimistic or
- optimistic

These options are applicable only if either the same Suffix Tree or the *Suspicion Suffix Tree* is used after the reconfiguration, as when using the approach to build up a Suffix Tree from scratch, all previous classification results are lost.

The *pessimistic* approach reflects the assumption to not continue the execution of a task that may probably lead to an unsafe system health state. Whenever a task's behavior sequences reaches a node in the Suffix Tree that is marked as *red/danger* (obtained in the previous configuration), its execution is aborted after having checked the classification marker value. Even if, in the new configuration, the concerned system call may not lead to cause a *red/danger* signal, there is a high probability that it will. Hence, it is considered as too risky to allow the task to continue its execution. This, in particular, is an option chosen for (safety or mission) critical applications.

In contrast, the *optimistic* approach relies on the assumption that after a task's reconfiguration the effects of behavior patterns on the health state may change. This means that a task is allowed to continue the execution of a behavior sequence at least once in the new configuration in order to evaluate the resulting health state. The realization of this option requires some extensions on the Suffix Tree and the workflow:

1. **Extension on the Suffix Tree**
   In order to distinguish whether a classification marker is associated with the previous or the current configuration, a further binary marker `classification_invalid` was added to the `TreeNode`.
   - `classification_invalid = 0`: classification marker is assigned to the current configuration
   - `classification_invalid = 1`: classification marker is assigned to the previous configuration

   The Anomaly Detection Module initiates that all nodes of the task's Suffix Tree obtain the value `classification_invalid = 1`. The marker is set to `classification_invalid = 0`, if the task's behavior sequence reaches the node in the new configuration.

2. **Extension of the workflow**:

   If a node's `classification_invalid` is `0`, the classification workflow is performed as defined without any deviation.

   If a node's `classification_invalid` is `1`, the checking of the classification marker value of the node is skipped, as, for the time being, it is not of interest concerning the current classification. This prevents the abortion of the execution in case the node was formerly classified as *red/danger*. However, the value of the classification marker is preserved as it is stored in the `previous_classification_marker`. The classification of the node is performed as defined by the classification method and finalized by setting the classification outcome into the node's classification marker. Before returning back, `classification_invalid` is set `0`.

   Preserving the former configuration's classification marker value becomes important in case of a unhealthy classification outcome in the current configuration. This information may be exploit for inducing a reaction as it may give hints to the source of the problem.

All these options how to deal with a reconfiguration are specific to the tasks' characteristics. The decision which strategy to choose requires knowledge concerning the tasks, their profiles, etc. The SCL Configuration allows the application designer to configure each task independently.

The inflammation signal is defined to signal a reconfiguration performed by the Controller. A reconfiguration not only changes the system or the applications configuration, but, furthermore it has impact on the procedure of the Anomaly Detection Module.

A change in the system configuration, in particular, has also effects on the parameters collected by the OS Health Monitor. Especially, parameters representing history and average values may become useless immediately after the reconfiguration. It is possible that strong deviations from the former values will be observed. Also deviations may be caused by an initialization phase that is performed after a system reconfiguration.

A similar situation is present at system start up: the Behavior Knowledge Base and the OS Health Monitor Database are empty. There is neither knowledge about the applications' behavior nor data about system performance and the associated health parameter. While initializing the system at start up, on the one hand strong deviations in the monitored data are expected in the first few periods, but on the other hand, parameter values can be different from those expected during the later system execution. For example, the initialization phase implements extra workload that is executed only once. This may lead to other values in the parameters of the Processor Utilization Monitor and the Scheduler Monitor, among others.

It will take some periods until enough values are monitored to establish reliable history and average value database that represents the current configuration's performance. With respect to the specification of the Signal Generator rules, those rules which operate on history or average data and are aimed to recognize deviations may lead to alerts in forms of *warning* or *danger* health signals if processed right after the reconfiguration or the system start up (which, in fact, is a new configuration). However, these health signals may be false in the context of the new configuration. Hence, it is essential to address this situation in order to prevent false alarms in the health signals.

To overcome this, the inflammation signal is integrated into the OS Health Monitor. Currently, the Anomaly Detection Module forwards the information in inflammation signal *Infl* to the

OS Health Monitor. This information can be exploited by the OS Health Monitor by means of integrating it to the Signal Generator rules. Different strategies can be implemented here:

1. deactivate the evaluation of designated Signal Generator rules for a fixed number of periods

2. incorporate the inflammation signal by means of weighting the parameters in the Signal rules; depending on the semantics of the parameters, some require a softening while others will have to be strengthened. For the impact of the inflammation signal several options can be selected:

   - continuous reduction of the weighting for a fixed number of periods
   - continuous reduction of the weighting until a stable range of values is achieved

Which strategy is the best one depends on the semantics of the parameters and the Signals Generator rules. This can only be chosen by specifically considering each single rule and its related parameters and, therefore, requires detailed expert knowledge. Especially, question about how many periods are effected by the inflammation signal and what are stable ranges of parameter values are very challenging and system specific.

We have realized and evaluated the impact of the inflammation signal on the Signal Generator rules by simply using a fixed value of periods in which designated rules are deactivated. However, with this strategy there is always the risk of missing some relevant *warning* or *danger* signals. Even though, the implementation of the second strategy is already prepared in the Anomaly Detection Framework, its realization requires a detailed investigation to specify the proper adaptations.

8.8.3 Learning

Referring to the requirements formulated on the Anomaly Detection Framework in Chapter 1.1, the capability of learning is one of the key attributes set on the Anomaly Detection Framework as it has to cope with the system dynamics in terms of changing behavior, changing system states and changes in the environment. Learning is implemented by the Anomaly Detection Framework on different levels:

Learning in the Knowledge Base

Learning is enforced by the system characteristics and is a means to cope with them. The following tasks are defined in conjunction with learning:

1. detect and collect new behavior

2. monitor the system health state and altering effects on the health state

3. explore *stable* or *normal* behavior, if it can be identified

The objective of learning connected to these tasks is collecting knowledge and its continuous updating. This knowledge is stored in the Behavior Knowledge Base of our Anomaly Detection Framework. To realize learning in the Behavior Knowledge Base, no additional effort was required, as the mechanisms implemented in components of the Anomaly Detection Framework implicitly implement learning:

1. **Suffix Tree**:

   New behavior patterns are immediately detected at the point of time they are matched against the Suffix Tree. A new behavior pattern is identified by the fact that no path for that pattern exists in the Suffix Tree. If the anomaly detection is executed online, then, the identification of the novel pattern happens online, namely exactly at the point of time the first system call occurs that does not match any path descending from the path that identifies the sequence build up until then.

   Whenever a system call is detected to be not yet existing as a node in the path of its coresponding sequence, it is directly included into the Suffix Tree by the *Searching and Extension* procedure introduced in Section 8.5.5. By this automatism, the Suffix Tree is extended with novel patterns as soon as they occur which, consequently, implicitly implements learning of behavior in the Behavior Knowledge Base.

   The existence of a behavior pattern (whether new or known) in the Knowledge Base is a prerequisite for its classification.

2. **OS Health Monitor**:

   The OS Health Monitor stores history data in its data base and, thereby, provides the basis of learning: history data allows to observe the evolution of the system's health parameters and the corresponding health state, and to derive trends and tendencies of the system state. Mainly, history data of the OS Health Monitor is used by the Signal Generator defining different classes of rules (see Section 8.6.4) that are related to learning:

   On the one hand, storing the history data of health parameters enables to examine whether the health parameters' data converges towards a particular value or state. The changing in values of health parameters is taken into account for Signal Generator rules classified as *Analysis of evolving data*.

   On the other hand, this data can be exploited as input for prediction methods (which are not implemented yet, but classified as Signal Generator rules based on *Observation data analysis*).

3. **Classification Marker**:

   The classification marker defines the obtained health state that was caused by executing the classification entity. Hence, the classification marker is a means to monitor the system health state in the context of behavior patterns.

   Storing the value of the previous classification marker in order to be able to compare it with the current classification outcome is maybe the simplest form of learning, if it is at all. Nevertheless, it enables to make statements and conclusions on health state changes. Furthermore, it is discussed in Section 8.7.3 and in Appendix F.1 how the previous classification marker value may contribute to the procedure of computing the current classification marker value.

4. **Occurrence Counter**:

   The Occurrence Counter integrated into the Suffix Tree's leaf nodes stores the number of occurrences of a sequence within the system's lifetime. The value of the Occurrence Counter provides information to distinguish between *common* behavior patterns called *normal* behavior - if it is high - and - if it is low - *uncommon* or *anomalous* (but not necessarily dangerous) behaviors.

Defining the ranges for that distinction is only possible with respect to a significant total number of executions of a task (its number of instances).

Even though the information in form of the Occurrence Counter is available, it requires further investigation to specify conditions for determining what is a significant number of executions. In this context, the question is still open which characteristics allow to declare a task's Behavior Knowledge Base as *stable*. Answering this question, however, is application specific and in in the context of this thesis will be left for future reserach.

The mechanisms implemented by the Anomaly Detection Framework allow continuous learning (in terms of novel behaviors, tendencies in behaviors, of their corresponding health states and its altering) at a basic stage. By making use of the information provided by the Anomaly Detection Framework, statements and conclusions can be made on the behavior and system health state right now. However, using the information in more sophisticated methods may enable more efficient conclusions or predictions.

Learning for Optimization

The Anomaly Detection Framework, as designed for self-x systems, is required to be self-adaptable in terms of the system characteristics. However, to comply with the entire system design, the behavior of the framework is expected to be self-optimizing as well.

Learning for optimization is implemented by the Anomaly Detection Framework in the Signal Generator rules. The rules belonging to the class of *Analysis of evolving data* and *Observation data analysis* execute (pattern) learning methods. Because of the system restrictions in terms of memory - especially related to the OS Health Monitor-, the data collected as history is bounded and small. The implemented learning methods, therefore, are designed to operate on these small data sets which are offered by the OS Health Monitor mainly reflecting the short term history. Examples for specifically applied learning methods are discussed in Section 8.6.4 which, predominantly, belong to the class of supervised methods (see Chapter 2.2.2).

Furthermore, numerous decisions in the anomaly detection are designed to be configured by thresholds (in the SCL Configuration) that have to be determined by expert knowledge. Learning algorithms, however, may achieve better results than human experts. Therefore, the Anomaly Detection Framework allows to exchange the threshold-based mechanism by other, more efficient methods, such as learning mechanisms. Integrating learning algorithms may optimize the effectiveness of the approach in terms of enhancing the classification precision and reducing false classification results. As these learning mechanisms are neither implemented nor evaluated yet, their implementation may become part of future research.

8.8.4 Forgetting

During the system operation, the Suffix Tree of an application tends to grow. Because the Suffix Tree is desired to store the entire behavior history of a task, it grows as long as novel behaviors are identified. The maximal size of a Suffix Tree is strongly depending on the characteristics of the application's behavior. For each configuration, its size (in terms of number of nodes) is bounded and determined by the method explained in Section 8.5.6.

However, the occurrence of behavior patterns, on the one hand, may be related to a specific life cycle phase of a task: the behavior during the start up is associated with the initialization phase and may lead to behavior patterns that might only occur at the initialization but will not occur during further operation again. Furthermore, a task may execute a particular behavior pattern only under a specific very rarely occurring condition. On the other hand, the behavior of a task can change and evolve over time. This can lead to the phenomenon that behavior patterns that have been common in the past will become uncommon. A specific case of such an evolvement in behavior may be produced by a task's reconfiguration. If the option was chosen to use the same Suffix Tree after the reconfiguration, there may exist behaviors that have been associated with the previous configuration but which may become obsolete for the current configuration. However, they are still kept in the Suffix Tree.

All these diverse situations illustrated above lead to a common consequence: after some time of system operation, the Suffix Tree of a task may contain paths that are irrelevant for the application's current life cycle phase. Concurrently, - because strongly restricted in embedded systems - these paths consume costly resources in terms of memory space.

One resulting problem is that memory is allocated that is of no avail. A further, and more critical problem arises if the Suffix Tree tends towards reaching its maximum size or even exceeding its specified memory (e.g. because of storing the entire history of numerous of task's configurations, then the maximum memory usage may become of exponential size). Because of the system dynamics, the operating system itself may even be in a critical state of memory usage. Then, a means to overcome facing such problems is to eliminate the previously mentioned irrelevant paths in order to gain free memory space. In context of our anomaly detection, we call this elimination the *Forgetting Process*.

The main idea of the *Forgetting Process* is, first, to identify those irrelevant paths, and, second, to delete them from the Suffix Tree. Three characteristics identify such paths:

1. the path represents safe/healthy behavior and

    1.1 the number of occurrence is negligible

    1.2 the behavior pattern is outdated as its last time of execution was long ago

Because of the importance and role of previously identified suspicious or dangerous behavior patterns, *red/dangerous* or *yellow/warning*-marked paths can never belong to the group of irrelevant paths. Therefore, the first characteristic specifies that only those paths classified as *green/healthy* can be defined as irrelevant paths because of their little effect on the entropy. This requirement is exclusive for defining a path as irrelevant.

The second characteristic is directly bounded to the Occurrence Counter that was already introduced. To implement the third characteristic, an extension of the leaf node by the attribute `visit_time` is required. Any visit of a leaf node not only increments the node's Occurrence Counter, but furthermore, it sets the attribute `visit_time` to the value of the system's current timer. By doing so each behavior pattern is marked by its last time of execution.

To identify the paths to be deleted from the Suffix Tree, the *Forgetting Process* evaluates for every leaf node of the Suffix Tree whether it identifies a path being irrelevant to the current application behavior. Therefore, it first checks the exclusive condition whether the classification marker is of *green/healthy* value. Only those nodes with a *green/healthy* classification marker are considered by the *Forgetting Process* further on. For them, two remaining conditions on the characterizing attributes are verified:

1. **occurrence counter below a predefined value**; the value is a threshold for a negligible occurrence. It is to be configured by the application designer that can set by either

    - a fixed absolute integer value - or by

- a dynamic threshold associated with a specified rate (e.g. 2 %). The absolute value for the threshold is computed on demand, based on the total number of executions of the task.

2. `visit_time` **is below a predefined value**; the value is a threshold for outdated behavior patterns. It is to be configured by the application designer that can be set either by

   - a fixed absolute integer value specifying time tics that define the time slot in which the path was not visited. The absolute value for the threshold is computed on demand by subtracting the specified value in time tics from the current timer value - or by
   - a dynamic threshold associated with a specified number of executed instances (e.g. 100). For the computation of the absolute threshold value, first, the time interval is statically computed based on the number of executed instances multiplied with the task's period length and, second, this time interval is subtracted on demand from the current timer value to achieve the threshold value.

   An additional option for `visit_time` threshold is the `configuration_startup`. The `configuration_startup` represents the time stamp of starting the execution of the current configuration (after a reconfiguration decision). If this option is chosen by the system designer, the threshold value is always set at the point of time of a reconfiguration, which in fact from the viewpoint of the anomaly detection is the arising of the inflammation signal.

If at least one of these two conditions is valid for a leaf node, a irrelevant path - ending at that leaf node- is identified and determined to be deleted from the Suffix Tree.

For deletion, it is necessary to identify the nodes that belong to the pattern determined by the irrelevant path as similar behavior patterns share their paths in the Suffix Tree. Therefore, the Suffix Tree is traversed backwards from the leaf node until either a branch node (last symbol of a shared paths) or the root node are reached. While traversing, all reached nodes - apart from the destination (root or branch node) - are deleted and their memory is freed.

The outcome of the *Forgetting Process* is a Suffix Tree that contains all previously identified suspicious and dangerous behavior sequences and all healthy *relevant* behavior sequences. In fact, it can be considered as a kind of garbage collection.

The system designer is allowed to decide in the SCL Configuration whether to activate and to allow the *Forgetting Process* to be included into the Anomaly Detection Module. Obviously, cleaning up the Suffix Tree is a comfortable feature, but at the same time it introduces further complexity and runtime overhead into the system. If chosen to be activated, the *Forgetting Process* on the one hand must be triggered by the Anomaly Detection Module and scheduled by the operating system (at least taken into account in terms of the Schedulability Analysis).

Triggering the Forgetting Process

There are different causes to activate the *Forgetting Process* also being of different criticality such as the Suffix Tree has grown too large or the system might potentially run out of memory. We classify the causes for triggering the *Forgetting Process*:

1. **soft**

A *soft* cause includes no necessity of cleaning up the Suffix Tree. The *Forgetting Process* is executed for optimization purposes in order to reduce the complexity of the Suffix Tree but without any external enforcement. This *soft* triggering is processed in the system idle times by using Background Scheduling. Instead of the processor becoming idle (empty ready queue in the Scheduler), the Scheduler calls the Anomaly Detection Module to execute the *Forgetting Process*. The *Forgetting Process* is executed as long as no hard real-time tasks are pending. At the point of time, a task becomes ready, the *Forgetting Process* is being interrupted until the next idle time reactivates its execution.

2. **hard**

   In case of a *hard* cause for triggering the *Forgetting Process*, its execution is enforced. This enforcement may be caused by internal and external causes. An internal reason is that the Suffix Tree (nearly) reaches its specified size and potentially tends to exceed it. An external reason is that the system tends to run out of memory. In both cases, memory has to be freed and one means is executing the *Forgetting Process*.

   The Anomaly Detection Module internally monitors the size of the Suffix Tree and, therefore, is aware of their memory usage. In case of a potential problem, the Anomaly Detection Module is responsible for calling the *Forgetting Process*.

   In case of an external, generous system problem, the Anomaly Detection Module has to be pushed to trigger the *Forgetting Process*. Therefore, a method `clean_memory()` is specified that can be actively called by the system Controller or even the Memory Manager itself.

Runtime Complexity

For the discussion of the runtime overhead coming with the *Forgetting Process*, we again distinguish between *soft* and *hard* triggering causes. The runtime complexity in case of *soft* triggering can be ignored because it only utilizes idle times. This, in fact, has no effect on the remaining system performance.

In case of enforced *hard* triggering of the *Forgetting Process*, its execution might be of high priority as initiated because of a potential thread. While on the one hand its execution is intended to prevent a threat, on the other hand it introduces additional runtime load that interferes with the execution of the remaining system entities. In order to ensure the schedulability of the entire system, it is essential to evaluate the run-time overhead produced by the *Forgetting Process* to be aware of its worst-case run-time costs.

The basic ingredient that contributes to the run-time complexity is the deletion of a node which requires constant time. The transition to a node is also valued as constant as the parent nodes are allocated by a pointer. This constant factor is added to the constant time required for deletion. Hence, the run-time complexity is determined by the number of nodes to be deleted. This is defined by the number of nodes of the set of irrelevant paths. In worst case, all the paths in the Suffix Tree may be identified as irrelevant paths so that the run-time complexity of the *Forgetting Process* is determined by the size of the Suffix Tree derived in Section 8.5.6. This worst-case run-time complexity has to be reserved for the purpose of executing the *Forgetting Process*.

Even though it has to be reserved, in average case, the worst-case run-time complexity will not be used by the *Forgetting Process* as it is very improbable to induce deleting the entire Suffix Tree. There is also the question of whether it is always necessary to delete all the irrelevant paths or whether it is more efficient to only delete the amount of nodes needed to put back the system into a stable and healthy state. In that case, the *Forgetting Process* could be aborted if such a state is reached. This may have different effects on the run-time complexity of the *Forgetting Process*. As this alternative approach is not implemented yet, it could become a task for future work.

8.8.5 Dealing with Classification Outcome

The Anomaly Detection Module is responsible for performing the classification of the monitored application's behavior and stores all behavior patterns with its associated classification results in its Behavior Knowledge Base. It is understood as the part of the Analyzer in the self-x architecture of ORCOS (see Section 8.2). There are classification results (in cases of a dangerous system states or even in suspicious states) that require a reaction. The reaction of classification results is given over to the responsibility of the Controller (see Section 8.2.3). The Controller implements a decision making strategy that is dedicated to induce a reconfiguration of the system, a system component or a task if required because of a system instability.

As the implementation of a reaction is not in the responsibility of the Anomaly Detection Module, it is not part of this thesis. However, the classification outcome that is hold by the Behavior Knowledge Base serves as the basis for any activity of the Controller. Therefore, the Anomaly Detection Module is required to provide interfaces to hand over the classification results to the Controller. It is provided a method called `getSignals(SuffixTree* t)` that returns the classification marker values of the leaf nodes of the associated Suffix Tree.

In general, the Controller is usually implemented in ORCOS as a WorkerThread that is regularly activated to execute its implemented strategy. It uses the method `getSignals(SuffixTree* t)` to obtain its input from the Anomaly Detection Module. By use of the WorkerThread, the Controller is scheduled as any other system component which may lead to the fact that a reaction initiated by the Controller may be delayed. This may be sufficient for *warning* signals, however, in critical situations, - if a dangerous state is present -, an immediate reaction is necessary. In order to ensure this immediate reaction, the Anomaly Detection Module provides a further interface to the Controller: if a node in the Suffix Tree is classified as *red/dangerous*, the Anomaly Detection Module directly sends a signal called `dangerAlert` to the Controller to trigger its prompt execution.

The Controller's execution basically relies on the values of the classification markers which are preserved in the format as defined in Section 8.7.1 and configured by the system designer. Based on the configuration of the classification marker, in case of a suspicion or a threat, the Controller is able to extract information about the potential source of a problem. This could be:

- the task in general, if multitude of classification markers in the task's Suffix Tree show up warning signals
- a specific behavior sequence executed by the task
- a system component delivering a warning signal in any sequence that operates on this component.

Of course, the quality of detecting the source of the thread depends on the configuration of the classification marker, but also on the sensitivity of the decision and reaction strategy: whether only a danger signal will induce a reaction decision, or whether multiple or even only one warning signal/s may lead to a reconfiguration. All the decisions related to when to react and how to react are left to the responsibility of the designer of the Controller.

8.9 Summary

This chapter illustrates in detail the implementation of the Online Anomaly Detection approach proposed in Chapter 7 in the real-time operating system ORCOS. As a prerequisite to integrate the Online Anomaly Detection, ORCOS was extended to support Online Reconfigurability by means of the Profile Framework as well as its architecture was extended into a self-x architecture composed of a Monitor, an Analyzer and a Controller. The basic self-x architecture was then specified into the Online Anomaly Detection Framework composed of a System Call Monitor, an OS Health Monitor, and the Anomaly Detection Module responsible for holding the Behavior Knowledge Base and performing the Classification.

The System Call Monitor extracts the system call information from the System Call Manager at the point of time a system call is executed. The Behavior Knowledge Base is responsible to store the behavior sequences and is realized by Suffix Trees. In this chapter, it is introduced in detail how the Suffix Tree is build up for successively arriving system calls and how the Suffix Tree is extended by novel behavior sequences.

The health state of the system is monitored by the OS Health Monitor which is composed of distributed Monitor Modules related to the system components. By means of specified Signal Generator rules, the OS Health Monitor delivers the health signals as input for the Classification of the application behavior.

For classification purposes, the Behavior Knowledge Base holds additional information about the behavior sequences such as the Occurrence Counter and the Classification Marker. We show how the execution of a system call is set in context to the health state: every system call is assigned a set of effected health parameters that in turn are associated with Signal Generator rules defined in the OS Health Monitor. For every system call executed, the effected set of Signal Generator rules is processed and results into a health signal. This ensures an immediate classification of the application's behavior and the detection of suspicious or malicious behaviors in an online manner. The resulting health signals are stored into the Classification Markers of the nodes representing the system calls in the Suffix Tree. The Behavior Knowledge Base contains all information about application executions and their corresponding effects on the system health state.

We have discussed the runtime overhead produced by the components of the Anomaly Detection Framework in the according sections. Furthermore, we have formulated a worst case estimation for the runtime overhead of the overall approach. As a main result, the runtime overhead introduced by the Online Anomaly Detection extends the execution of each system call and is therefore a function of the worst case length of the behavior sequences of a task, resulting in $O(n_{max}^2)$. The runtime overhead strongly depends on the characteristics of the tasks intended to be executed on the system and therefore, must be integrated into the task's Schedulability Analysis.

Of course, we provide some ideas for further research and point out some questions that are still open.

CHAPTER 9

Evaluation of Costs

Chapter 8 illustrates adaptations made in ORCOS in order to transform ORCOS towards a real-time operating system for self-x environments, in the first place, and presents the implementation of the ORCOS modules that realize the Online Anomaly Detection approach.

Any extension of the ORCOS kernel leads to an extension in the footprint of ORCOS as well as it contributes to runtime overhead. Real-time systems, however, must guarantee full determinism in terms of resource usage and timing (timeliness). Any contribution to runtime or memory usage has to be taken into account in the evaluation of resource usage and schedulability analysis of the system.Therefore, measuring any overhead produced by any system (part) extension is essential for real-time systems which in our context implies measuring the overhead produced by the Online Anomaly Detection. This can only be obtained by taking measurements of the system in execution.

The modules that compose the Online Anomaly Detection are implemented as encapsulated system entities which allows to execute them in isolation in order to perform stand-alone quantitative measurements. In this chapter, we present the results of runtime overheads and memory space requirements obtained in the evaluations of the individual modules of the Online Anomaly Detection.

The evaluations have been encapsulated within isolated work packages (mostly master theses). Hence, the evaluations have been performed on different architectures, platforms as well as environments, but always in the context of ORCOS. The most relevant evaluations have been predominantly performed on a RAPTOR2000[1] board with a PowerPC405 200 MHz processor, and within the PowerPC405 emulator QEMU[2](Version 1.5.0) with 300 MHz. For any module evaluation, the applied evaluation platform is explicitly mentioned.

Specific Remarks to Evaluation Results under QEMU

QEMU is an emulator that supports user-defined hardware platforms, such as for PowerPC405. QEMU uses the dynamic translation technology to enable good emulation speeds as it aims for fast performance evaluations e.g. for rapid prototyping purposes. Even if supporting embedded platforms, QEMU, however, does not support cycle-accurate simulations. The fact

[1] RAPTOR2000 was developed at the group of Prof. Rückert at the Heinz Nixdorf Institute
[2] www.qemu.org

that QEMU itself is multithreaded and executed on PC operating systems such as Linux or Windows that are multitasking- and threaded themselves, builds in many factors of inaccurate behavior and, therefore, makes cycle-accurateness impossible.

In order to increase the accurateness, the QEMU timer (a fixed interval timer) was exchanged by an implementation of the programmable interval timer that acts as the clock generator (also by suspending the timer if ORCOS is suspended) and controls the frequency of the ORCOS timer [69]. This does provide more reliability in the obtained time measures, but still does not approach cycle-accurateness.

Nevertheless, we have conducted some significant measurements in the context of the evaluation of the Online Anomaly Detection under QEMU and ignored QEMU's lack of cycle-accurateness because of the following reasons:

Before an implementation comes into operation on a real-time system, it has to pass static analysis in order to ensure its predictability, meaning its boundedness in terms of runtime. Hence, evidence of the predictability and runtime complexity of an approach is provided before the system is set into execution. The time measurements performed at runtime are not dedicated to proof the timely accurate behavior, so that we are not really interested in the delivered absolute values. Furthermore, the entire ORCOS execution suffers from the dynamics of the underlying emulator so that the measured execution times are distorted by an approximately equal order of magnitude. Based on this assumption, the accuracy of timing measures is sufficient enough as only relative results are of interest: With the timing results, we are able to assess the runtime overhead in relation to the remaining operating system performance as well as to conclude tendencies in runtime behavior based on the evaluation results. (In this thesis, we are more interested in showing the effectiveness of the Online Anomaly Detection, see Case Study in Chapter 11, rather than in detailed assessment of the costs of the approach.)

The second motivation to use QEMU for the evaluations is convenience: In the first place, ORCOS executing under QEMU is easier to be debugged than when executing on the RAPTOR2000. Runtime data can be easily gained from the debugging interface. Furthermore, ORCOS that runs under QEMU on a host system has access to the host file system so that (by the aid of some scripts) runtime data can be easily written into log files to be analyzed after the execution.

The last but not least reason to apply QEMU is that by using a software emulator, we have full control over the hardware. This leads, on the one hand, to the ability to manipulate the hardware configuration according to the scenario requirements (e.g. equip the PowerPC405 with more memory if required by the evaluation use case of the Online Anomaly Detection). On the other hand, full control allows manipulations in terms of fault injections that are of importance for verifying the effectiveness of an anomaly detection approach.

9.1 Evaluation of the System Call Monitor

The System Call Monitor extracts the required system call data at the point of time the system call is executed. This, of course, on the one hand requires memory available for storing the extracted data and it increases the runtime of the execution of a system call. We have evaluated both, the memory consumption and the runtime overhead, produced by the System Call Monitor. The here presented evaluation results refer to experiments performed in the context of the Master Thesis by Gavin Vaz [92]. Here, we only present the main and most relevant results. More sophisticated discussion on the evaluation of the System Call Monitoring

Framework including more sophisticated results are provided in [92].

Monitor Buffer Configuration

A major decision at compile time is the strategy for buffer allocation. Static buffer configuration leads to constant time in storing the system call-related data. This, of course, increases the runtime overhead of a system call, but only by a constant value so that the entire runtime overhead still remains bounded.

The more interesting question is concerned with dynamic buffer configuration. Even if there are approaches that realize dynamic memory allocation in real-time systems in constant time as in [75], up to now ORCOS does not implement such memory management strategies. In order to guarantee predictability of the system, the entire system behavior requires to be deterministic which implies the memory allocation method as well. Hence, it is essential to verify whether the available ORCOS Memory Managers are able to fulfill this requirement.

Therefore, the dynamic buffer allocation was evaluated on the *Linear Memory Manager* and the *Sequential Fit Memory Manager* with the following results:

1. *Linear Memory Manager*:

   The time costs for memory allocation with the *Linear Memory Manager* can be considered as nearly constant as illustrated in Fig. 9.1. Nevertheless, the *Linear Memory Manager* does not support any `free`- or `delete`-function. This is critical especially for systems that continually allocate memory for (monitoring) data like the System Call Monitor as at a particular point of execution the memory will become fully utilized which will make the system inoperative.

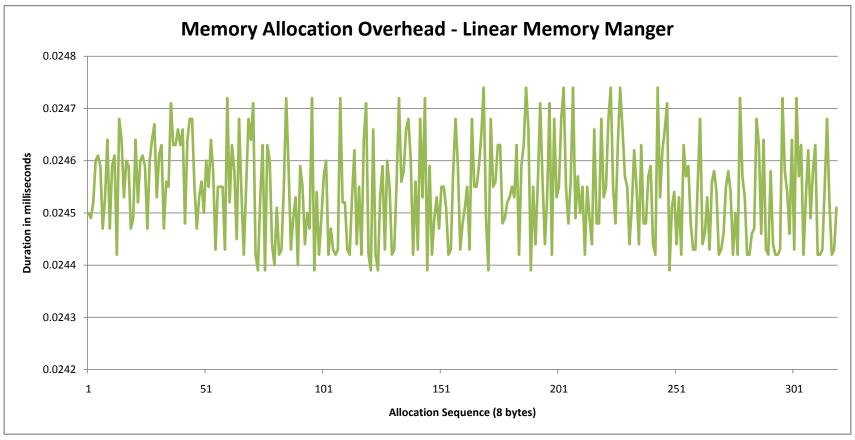


Figure 9.1: Time Overhead for Memory Allocation with *Linear Memory Manager* (source: [92])

2. *Sequential Fit Memory Manager*:

   The *Sequential Fit Memory Manager* offers `free`- or `delete`-function, so the risk to run out of memory is much smaller, as unused memory can be released. Because of this, free memory is not located contiguously but, moreover, can be spread over the entire memory space. The experiments in [92] have shown, that, by successively allocating memory, for any new memory request the time the Memory Manager requires for allocation tends to

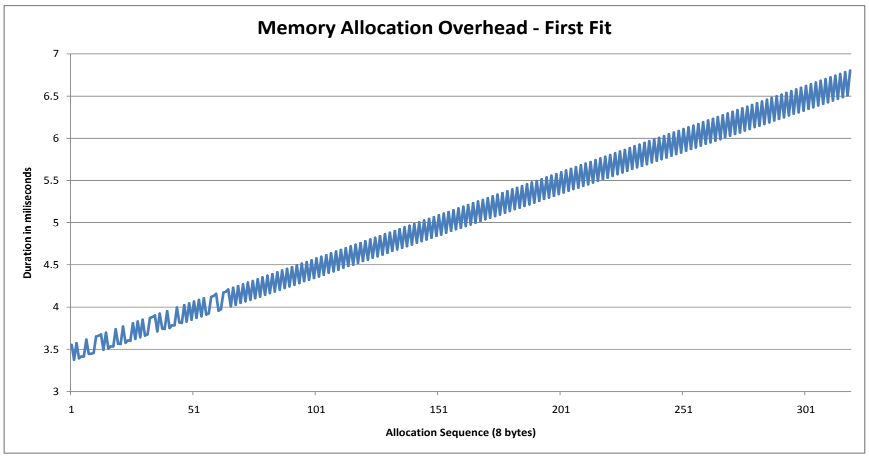


Figure 9.2: Time Overhead for Memory Allocation with *Sequential Fit Memory Manager* (source: [92])

increase. The results are illustrated in 9.2. The *Sequential Fit Memory Manager* becomes impractical with respect to the predictability of the system.

Based on these evaluations, dynamic buffer allocation is not applicable with the currently available Memory Management implementations in ORCOS. Hence, for the evaluation of the Online Anomaly Detection, static buffer configuration was configured for the System Call Monitor.

Space Overhead

The System Call Monitor causes memory usage with regard to two aspects which both have to be evaluated: first, the memory amount required by the Monitor Database and second, the memory space occupied by the System Call Monitor's code in the kernel's binary.

1. **Monitor Database Memory Overhead**

   The size of the Monitor Database is composed of the *buffersize* of a task multiplied with the size of the System Call Record summed up over all n tasks:

$$sizeOf(MonitorDatabase) = \sum_{i=1}^{n}(buffersize_i \cdot sizeOf(SCR)) \tag{9.1}$$

   If all tasks have equal *buffersize*, which is true in particular for static buffer configuration, then the resulting size of the Monitor Database is:

$$sizeOf(MonitorDatabase) = |J| \cdot buffersize \cdot sizeOf(SCR)) \tag{9.2}$$

   with J being the task set and $|J| = n$. Both, n and *buffersize*, are related to the settings in the system's SCL Configuration.

Despite the fact that the dynamic buffer configuration is impractical in terms of time predictability, the main difference in terms of space overhead between static and dynamic buffer allocation is the size of the System Call Record. While the decision between static and dynamic buffer allocation is made offline at compile time, the mode of the System Call Monitor is set (and can be changed) at runtime.

In static buffer configuration, in order to be able to support all the offered modes, the SCR preserves memory space for the entire set of fields required by the union set of modes. Hence, the size of the System Call Record is constant. On the contrary, when dynamic buffer allocation is applied, the SCR consists of only those fields required by the configured mode. Hence, the size of the SCR is varying.

Regarding the mode, the size of the SCR, furthermore, depends on

- the size of the return address stack, (based on the depth of the calling function, which was evaluated to be maximum 4 for the currently existing implementation)
- the number of system call arguments, (maximum number of system call arguments among the existing system calls is 5) and
- the number of acquired resources (maximum number of 6)

The resulting upper bound for memory size of the SCR is 76 bytes:

SCR Elements	Size
Basic Parameters: Thread ID, System Call ID, Time Stamp	16 bytes
System Call Arguments (4 bytes per each pointer to argument, with maximal 5 arguments)	20 bytes
Return Address Stack (4 bytes per entry, with maximal depth of 4)	16 bytes
Acquired Resources (4 bytes per entry, maximal 6)	24 bytes

2. **Kernel Binary Size Overhead**

   Depending on the buffer configuration in SCL, different parts of source code are compiled into the kernel. Hence, the size of the kernel binary is related to the buffer configuration:

Kernel Binary Version	Size
ORCOS Kernel System Cal Monitor not configured	657 KB
ORCOS Kernel with System Call Monitor static buffer configuration	853 KB
ORCOS Kernel with System Call Monitor dynamic buffer configuration	837 KB

The System Call Monitor increases the size of the kernel image by ca. 30% which is quite a large overhead. Mainly, this overhead is caused by the size of the lookup table used in the return address stack generation. If the monitoring parameters would be configured offline, the compiling source code could pertain only those parts related to the used parameters. This, in turn, would decrease the kernel binary size including the System Call Monitor.

Runtime Overhead

Apart from the predictability of the memory allocation, the predictability of the entire System Call Monitor performance has to be guaranteed. For the System Call Monitor, this is concerned with evaluating the overhead produced by extracting the system call information within a system call execution. This overhead was measured for the different system calls as well as for the different modes per system call. To ensure a certain reliability of the achieved values, the measurements have been replicated for 100 times per each configuration.

In the context of the Online Anomaly Detection, we are mainly interested in the extraction of the system call identifier. This is reflected in the mode of the basic parameters leading to a SCR that consists of Thread ID, the System Call ID and the Time Stamp. Hence, only the results related to the basic parameter mode will be discussed here. For taking a look at the performance parameters of other modes, see [92].

For every system call, the same data set is extracted. The experiments have shown that the required time for the extraction is nearly constant for all the different system calls which approves the boundedness of the System Call Manager. On the 200 MHz PowerPC405, the system call extraction takes in average case $14.74\mu s$ and $14.85\mu s$ in worst case which are 2948 timer cycles in the average case and 2970 timer cycles in worst case.

As the individual system calls have their individual runtimes, the measured time overhead extends each system call runtime by an additional amount. For example, `free` has the runtime of $6.09875ms$ leading to a runtime of $6.11327ms$ with system call monitoring which results in an increase of 0.24%. In comparison, for `malloc` which has a pure runtime of $6.33054ms$ and $6.34244ms$ with system call monitoring, the increase is 0.19%.

Nevertheless, we can conclude that System Call Monitoring, as it extends the runtime by a constant value per system call, is independent of the type of system call, and, therefore, only proportional to the number of executed system calls. This result is the best prerequisite for our Online Anomaly Detection, as the runtime for obtaining the system call sequences is bounded by the worst case length of the system call sequences.

9.2 Evaluation of OS Health Monitor

The OS Health Monitor collects and stores the health data delivered by the Component Monitor Modules and is responsible for calculating the health signals. The collection of the health data and the calculation of the health signals requires execution time while storing the according data produces memory consumption. Both, the runtime overhead as well as the memory requirement, have been evaluated in the work of Sijia Li in [69]. The experiments performed in that evaluations output quantitative results specifying the produced overhead for two different scenarios:

- The first scenario is specified as *low workload* as in this scenario 3 tasks are executed that do not produce much workload for the system. Two of the three tasks merely do nothing else than calling `print "Hello Task"` while one task is sending `"Hello Task"` messages via TCP to a remote server. This scenario was basically designed in order to have a stable system for the evaluation of data gathering as, because of the tasks' characteristics, the system is expected to remain in a *healthy state* (which was evidenced in [69]).
- The second scenario is specified as *high workload*. It is based on the execution of a *real* real-time application that was used for our case study: an application to control and

drive a small BeBot robot through a maze (see Chapter 11 for more details). As such an application is much more complex than a 'calling `print "Hello Task"`'-application, it produces much more workload in the system.

Because of the different characteristics of the scenarios, they provide best conditions for comparing different workload applications' evaluation results.

Instead of being triggered in an online manner related to a system call execution, for this evaluation, the OS Health Monitor was executed as a Workerthread. The main difference is that in the online manner only the respective *health data* is updated that is effected by the executed system call which leads to the collection of a (small) subset of the *health data* as well as a reduced subset of Signal Generator rules to be processed. In contrast, being executed as a Workerthread, the Health Monitor Center periodically triggers all Component Monitor Modules to collect the *health data* and generate the according *health signal*. This, in fact, represents the worst-case execution of the OS Health Monitor, as all *health paramters* have to be collected as well as the processing of all the implemented Signal Generator rules. Based on this, the following evaluation results offer upper bounds for the OS Health Monitor.

9.2.1 Memory Overhead

The *health data* is hold by the OS Health Monitor in a Monitor Database. The Component Monitor Modules collect, component-related health parameters as well as health parameters that are kernel-related and task-related. While each Component Monitor Module and the kernel only exist once in the system, the size of the parameter set is constant (because of this, the memory usage of component-related and kernel-related parameters are summed up to *Global Parameter of Component*). However, task-related parameters are assigned to each existing task and, therefore, the size of the parameter set is varying with respect to the number of tasks and their characteristics. For every task, the Component Monitor Modules are activated individually (in the SCL Configuration) in dependence of the system calls the task executes. For example, if a task does not use communication-related system calls, the Communication Monitor shall be deactivated.

The memory usage of the OS Health Monitor Database, based on the health parameters specified up to now, is illustrated in Table 9.1.

The history size is configurable by the system designer (by default it is set to $history\_size = 4$). The resulting memory size of the OS Health Monitor Database is:

$$sizeOf(OSHealthMonitorDatabase) = history\_size \cdot (348 \text{ byte } + n \cdot 976 \text{ byte })$$

For the two scenarios described above and a history size of 4, we obtained an OS Health Monitor Database size of (in case all Component Monitor Modules are activated for the tasks):

- *low workload* with 3 tasks:
  $sizeOf(lowWorkloadDatabase) = 4 \cdot (348 \text{ byte } + 3 \cdot 976 \text{ byte}) = 13104 \text{ byte } = 12.8 \text{ KByte}$
- *high workload* with 1 tasks:
  $sizeOf(HighWorkloadDatabase) = 4 \cdot (348 \text{ byte } + 976 \text{ byte}) = 5296 \text{ byte } = 5.17 \text{ KByte}$

Of course, the main question in this context is the number of history data required for obtaining a reliable *health signal* that has to be answered by data analysis experts. It is necessary to emphasize here that for the BeBot an IR Sensor Monitor was specifically designed (see

Global Health Parameters of Component	Size (in Bytes)
Communication Monitor	96
Memory Monitor	44
Processor Utilization Monitor	120
Scheduler Monitor	32
Device Driver Monitor	32
File Manager Monitor	24
Sum of Global Health Parameters' Size	348
Task-related Health Parameters	Size (in Bytes)
Communication Monitor	128
Memory Monitor	80
Processor Utilization Monitor	160
Scheduler Monitor	416
Device Driver Monitor	32
File Manager Monitor	160
Sum of Task-related Health Parameters' Size	976

Table 9.1: Overview of Memory Usage of Health Parameters

Chapter 11.2.5) is not part of this overhead evaluation. Obviously, its integration into the OS Health Monitor will enhance the size of the Monitor Database. For each IR sensor value, 4 Bytes are reserved in the Monitor Database, leading to $4 \cdot 12$ Bytes to obtain the entire set of distance values of the BeBot's IR sensors. With respect to the frequency of execution of those system calls that induce a change in the IR sensor values, the size of stored history data must be adjusted according to the period of the OS Health Monitor's execution to ensure no loss of state information.

9.2.2 Runtime Overhead

The two scenarios defining different workloads (described above) build up the basis for the runtime evaluations of the OS Health Monitor. Because the second scenario is only executable in the context of the case study - which requires ORCOS to run in the emulator QEMU - both, the *low workload* and the *high workload* scenario have been conducted on QEMU in order to make the results comparable. To obtain performance measures of OS Health Monitor exclusively - without any interference from the remaining components of the Anomaly Detection Framework - the OS Health Monitor was evaluated in a stand-alone manner, meaning that system call monitoring as well as the classification part of the approach have been deactivated. Implemented in form of a Workerthread the OS Health Monitor is regularly activated with respect to its period. ORCOS is configured to use the Earliest Deadline First Scheduler.

Some assumptions on the system behavior can be made based on the characteristics of the two application scenarios:

- **Amount of health data generated** In the *low workload* scenario, the three tasks are simple in terms of the number of executed system calls. While two of the tasks only execute the `printf` system call, the third one is additionally sending the TCP messages. Basically, the complexity of the task's behavior in terms of system calls governs the amount of health data generated. Therefore, the *low workload* scenario is expected to generate a small amount of health data.

In contrast to that, the application in the *high workload* scenario (see Chapter 11) uses much more system calls (mainly related to the BeBot's control, see Appendix A.10) and, additionally, requires the collection of *health data* from the IR Sensor Monitor. This generates much more health data as compared to the *low workload* scenario.

- **Diversity of health data** Irrespective of the characteristics of any task, a system executing (if the OS Health Monitor is activated) generates health data for the Scheduler Monitor and the Processor Utilization Monitor by default. Additionally, it generates the health data related to the OS components it employs.

  In the *low workload* scenario, health data for the File Manager Monitor (as `printf` are considered as file access) and Communication Monitor (as one task is using the Communication Module) is generated. The *high workload* application also generates health data for the Communication Monitor. However, as each IR Sensor request includes a communication over TCP, it generates much more data for the Communication Monitor than the simple *low workload* scenario. Furthermore, the *high workload* application operates on the IR Sensors which consequently leads to a generation of health data for the 'IR Sensor'-specific Device Driver Monitor implementation.

Of course, the amount of health data generated influences the execution time of the OS Health Monitor as it is responsible for its collection.

Apart from the amount of health data, the set of effected Component Monitor Modules features Signal Generator rules that also differ with respect to the number of specified rules and the complexity of calculating the respective health signals that lead to deviations in execution times.

To measure the OS Health Monitor's execution time, the Workerthread executing the OS Health Monitor Center was built as an object to be monitored by the Scheduler Monitor Module. The Scheduler Monitor Module collects the health parameters of the OS Health Monitor Center's execution and provides the data for the evaluation. Multiple diverse configurations have been set up for the experiments to measure runtime overhead of the OS Health Monitor. We discuss some significant samples reflecting typical characteristics observed in the experiments:

1. **Low Workload Scenario**

   In the *low workload* scenario, $Task_1$ is the task executing the communication over TCP. $Task_2$ and $Task_3$ are the tasks executing the `printf` system calls. The tasks have been executed with periods as specified in the table and exhibited the following summarized execution times (values obtained out of over 220 periods of execution):

Task	Period	average	best case	worst case
$Task_1$	$150000\mu s$	$85.43\mu s$	$52.93\mu s$	$284.08\mu s$
$Task_2$	$100000\mu s$	$69.06\mu s$	$32.57\mu s$	$235.29\mu s$
$Task_3$	$100000\mu s$	$64.97\mu s$	$31.16\mu s$	$217.11\mu s$

   The execution times of the tasks do not have any significant impact on our evaluation and are therefore consolidated here. An example illustrating the execution times of a task is provided in Fig. 9.3. Fig.9.3 clearly shows that the worst case execution times are outliers.

Reasons for these outliers have not been analyzed in the context of this thesis. We assume that these outliers are mostly caused by the dynamics of the underlying system QEMU and/or the nondeterministic behavior of the communication device under QEMU.

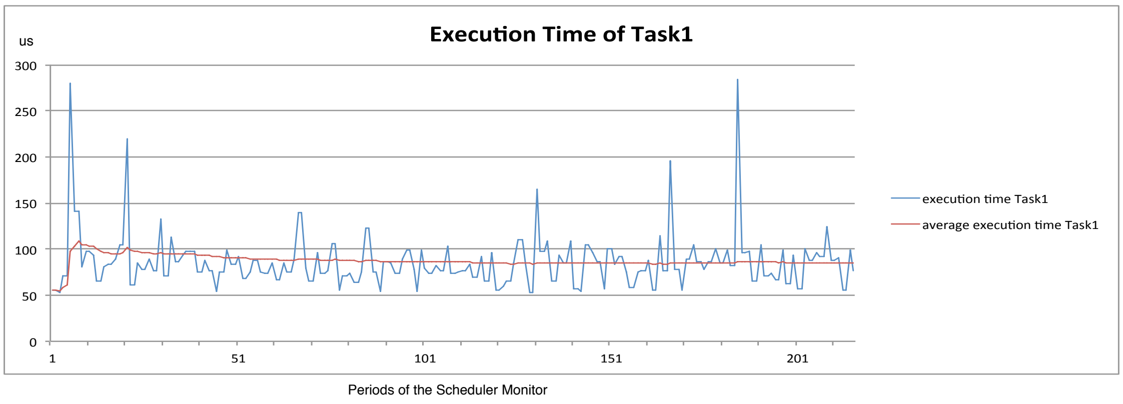
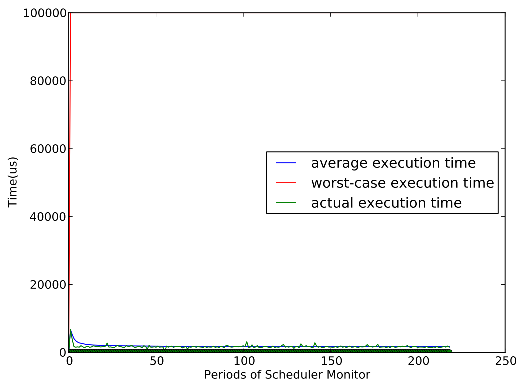


Figure 9.3: Execution Times of $Task_1$ in the Low Workload Scenario

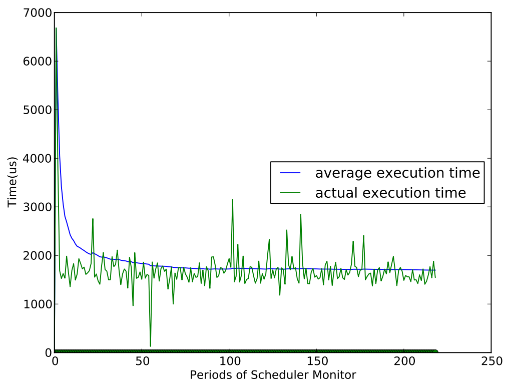
The OS Health Monitor was executed with a period of 100000μs. In order to prevent a critical situation for execution time that reaches the statically defined worst-case execution time, we have set the static worst-case execution time equal to the period. The measured execution times for the OS Health Monitor are illustrated in Fig. 9.4a and in Fig. 9.4b, both showing the same evaluation using a different resolution.

Basically, the execution times of the OS Health Monitor Center are in majority within the range of 1300μs and 1900μs, leading to an average execution time of ca. 1700μs. Referring to Fig. 9.4b, the number of outliers in execution time is small. Executing this scenario, the OS Health Monitor consumes mostly about 1% of the processor utilization (see Fig. 9.4c), apart from the outlier values.

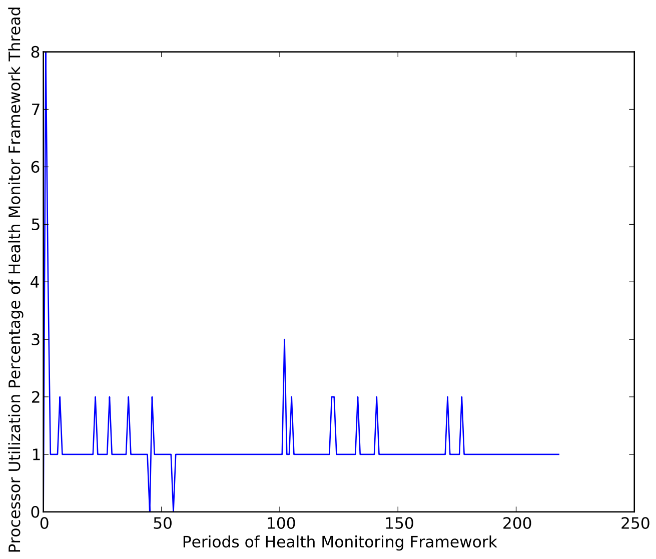
A striking point is the first period of the OS Health Monitor: the execution time of the OS Health Monitor is much higher than in all the following periods. This is due to the fact that the OS Health Monitor executes an initialization in its first period of execution that, obviously, requires more computation time.



(a) Execution Times of OS Health Monitor in the Low Workload Scenario measured by the Scheduler Monitor Module



(b) Focus on Execution Times of OS Health Monitor in the Low Workload Scenario measured by the Scheduler Monitor Module



(c) Processor Utilization by the OS Health Monitor

Figure 9.4: Runtime Performance of the OS Health Monitor for Low Workload Scenario

2. **High Workload Scenario**

   In this evaluation scenario, the application task (BeBot application) is configured with a period of $110000\mu s$ while the period of the OS Health Monitor remains $100000\mu s$. The execution times for the task are recorded in Fig. 9.5 over 150 periods of the Scheduler Monitor. Most of the values are in the range of $40000\mu s$ ($40ms$) and $60000\mu s$ ($60ms$), leading to an average value of ca. $49000\mu s$ ($49ms$) that reflects a processor utilization of about 45%. Again, the strong deviations in the collected execution times of the BeBot application could have been caused by the dynamics of QEMU and the nondeterminism of the Communication Device (this has not been examined further).

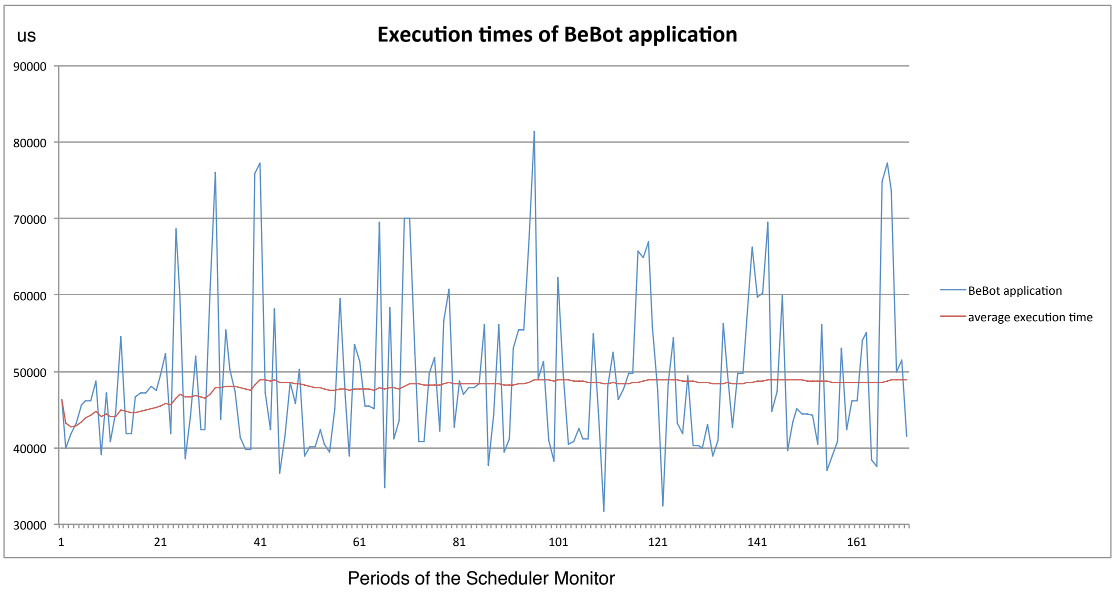


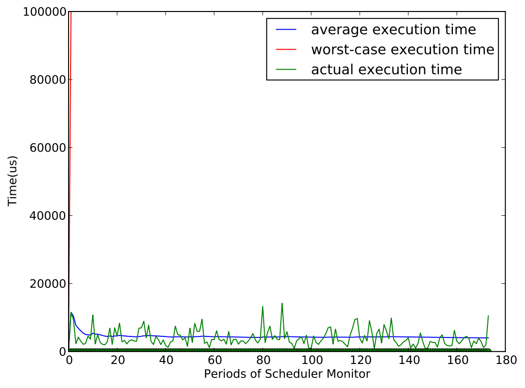
Figure 9.5: Execution Times of the BeBot application

   The according execution times of the OS Health Monitor are illustrated in Fig. 9.6a (with a more detailed view provided in Fig. 9.6b). The main range of the execution times is between $2000\mu s$ and $7000\mu s$, resulting in an average value of about $4000\mu s$ with strong deviation between the values. The processor utilization is depicted in Fig. 9.6c. Based on of the strong deviation in the execution times, the processor utilization shows up analogous deviations between 2% and 10% with some outliers of even 14%.

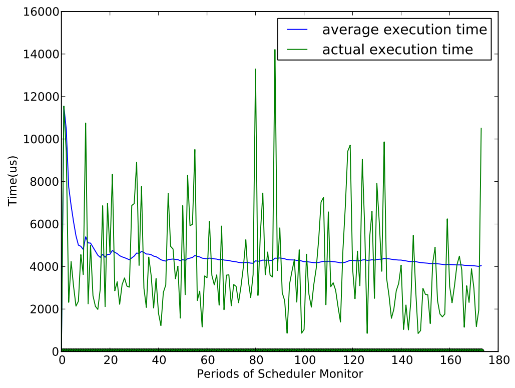
   In comparison to the *low workload* scenario, the OS Health Monitor requires much more execution time in this evaluation scenario. This is basically caused by the complexity of the application that generates much more health data needed to be collected and examined by the Signal Generator.

The results show the OS Health Monitor runtime overhead in a stand-alone evaluation. As already indicated, it is the worst-case execution of the OS Health Monitor: collecting and analyzing all health data of all Component Monitor Modules at once (instead of only a subset of only the effected health parameters if performed in an online manner).

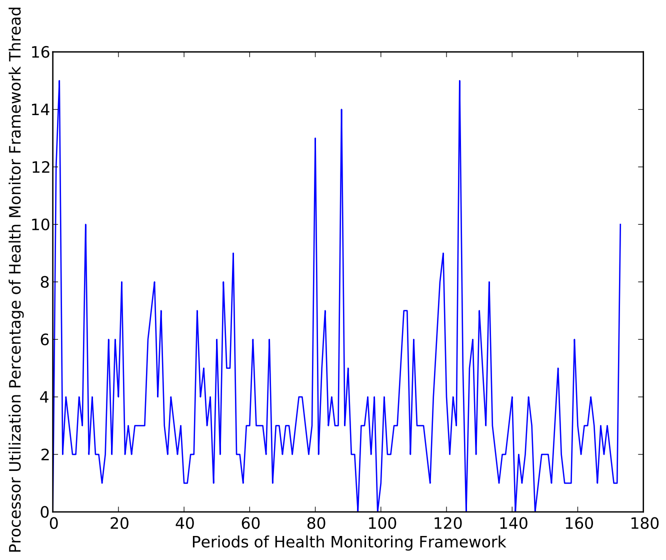
In contrast to this evaluation, in the case study, the OS Health Monitor was operating in the online manner in order to guarantee an online classification with health signals reflecting the current state on demand (according to the Online Anomaly Detection runtime process specification, see the Sequence Diagram in Fig. 8.33).



(a) Execution Time of the OS Health Monitor Center observed by the Scheduler Monitor Module



(b) Execution Time of the OS Health Monitor Center observed by the Scheduler Monitor Module



(c) Percentage

Figure 9.6: Runtime Performance of the OS Health Monitor for High Workload Scenario

9.3 Evaluation of Behavior Knowledge Base

The system calls extracted by the System Call Monitor constitute the system call sequences that are conserved in the Behavior Knowledge Base, in particular in the Suffix Trees. Including the system call data into the Suffix Tree is performed immediately before the system call's execution which consequently increases the runtime overhead of each system call. Furthermore, any extension of the Suffix Tree by new system call sequences enhances the size of the Suffix Tree. The results of the quantitive evaluation of the Behavior Knowledge Base including the runtime overhead and the memory space requirements produced by processing the system calls sequences have been published in [Stahl and Rammig, 2015].

In order to conserve the system call sequence into the Behavior Knowledge Base, first, it is verified whether the current system call ID exists as a symbol in the actual sequence path of the Suffix Tree. If it exists, the control is returned back to the System Call Manager to enable the processing of the system call. If the symbol currently is not included in the sequence path of the Suffix Tree, the current path of Suffix Tree is extended by that symbol (and the according paths of its suffixes) before returning back to the System Call Manager.

We have measured that overhead with some basic experiments in the context of the case study (see Chapter 11). For the evaluation, we used the *high workload* scenario defined in the section above with our autonomous robot BeBot that is run by ORCOS to drive through a labyrinth in order to find a dedicated destination. The BeBot provides several different algorithms that implement destination-finding strategies that can be selected or reconfigured online by the BeBot based on a specified optimization function.

For evaluation, we use our Virtual Reality (VR) environment introduced in Chapter 11 that integrates the virtual counterpart of the BeBot roboter which was modeled based on a real device. ORCOS is executed under QEMU (device emulator) on a 300 MHz PowerPC405 that is connected to control the virtual BeBot.

To obtain timing values, we preferred to select an application task that implements a simple (nearly linear) control flow in terms of system calls. The selected application realizes a simple strategy to just simply follow the right side wall in the maze. Its system call-based control flow is illustrated by Figure 9.7, containing the according system call IDs in the building blocks. It consists of 9 system calls in best case and a variation of 10 system calls in worst case (concerning the number of executed system calls). Because of the nearly linear control flow, we expect the major construction activity of the knowledge base to be fulfilled in the first period. In all subsequent periods, mainly matching activity shall be performed besides some exceptions in case of executing variants of the control flow for the first time.

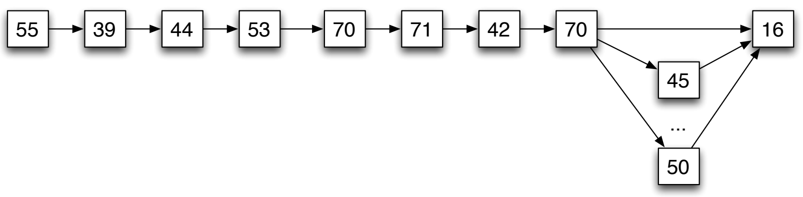


Figure 9.7: Control Flow of Example Application

In our Suffix Tree implementation, we have realized the Suffix Tree in forms of a suffix trie where each symbol is represented by one node. We have done this in order to simplify the process of Suffix Tree extension and for the purpose of avoiding the splitting of edge labels (that might be time consuming) at run time. The size of a node is 44 byte (including all required data, pointers to succeeding nodes as well as classification attributes). For our example (executing between 9 and 10 system calls), the complete Suffix Tree consists of 152 nodes which leads to a memory usage of 6696 bytes (6.54 KByte). The number of nodes in the tree (which in fact represents the number of label symbols in the real Suffix Tree) is caused by the structure of the system call sequence that does not contain any recurring subsequences (besides the recurrence of the system call having $ID = 70$). Of course, the size of the Suffix Tree is strongly governed by the applications running on the system (as discussed in Chapter 8.5.6).

Measuring the time overhead that is put on each executed system call we have to distinguish between two cases:

1. node already exists in Suffix Tree
2. node has to be extended into the Suffix Tree

The latter case of extending automatically includes the previously performed checking of whether the node already exists in the Suffix Tree (in terms of the current sequence path). Therefore, the effort required for matching is at least added to each system call execution when the anomaly detection framework comes into operation. Figure 9.8 shows the minimal overhead of each system call required for matching activity.

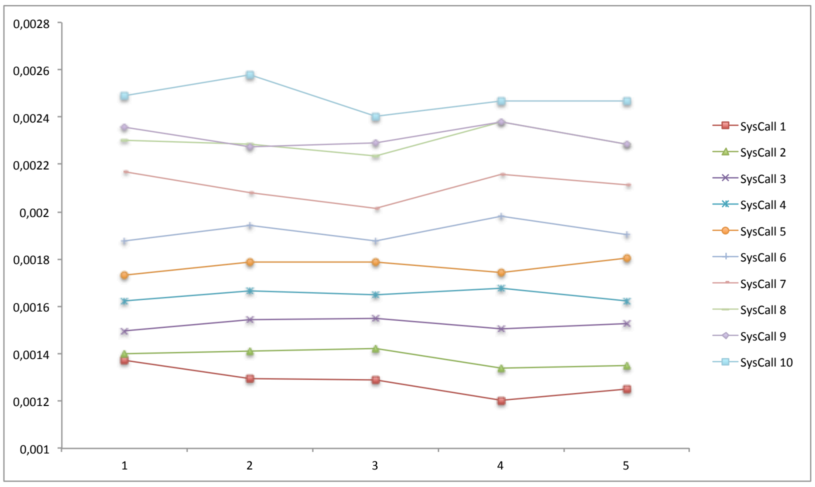


Figure 9.8: Runtime Overhead for extension of the Suffix Tree

From the concept of Suffix Trees, a symbol at the *i*th position in a sequence is contained *i* times in the suffix tree. Hence, the time for adding a system is expected to be related to its position in the sequence. This was proven by the measures shown in Figure 9.8: for the Suffix Tree extension, the time overhead increases with the position of the system call in the sequence. The higher the position of the symbol in the sequence, the longer the time required for verifying whether the symbol is already contained in the Suffix Tree, being $0,00128ms$ ($= 385$ timer cycles on a 300 MHz processor) for the first symbol in the sequence (as mean best case value) up to $0,0026ms (= 780$ timer cycles) for the 10th symbol in the behavioral sequence.

A summary of the execution times measured is provided in the following table:

position of system call	best case (in *ms*)	average case (in *ms*)	worst case time overhead (in *ms*)
1	0.001282667	0.005225062	0.513289333
2	0.001384667	0.004905975	0.287340667
3	0.001524667	0.005391915	0.381678
4	0.001646667	0.007031669	0.665592667
5	0.00177	0.00841225	1.014724
6	0.001915333	0.007239755	0.734574
7	0.002106667	0.010336985	1.297471333
8	0.002318667	0.01061829	1.337451333
9	0.002482	0.009894956	0.854320667
10	0.002591333	0.01247708	1.258096

For worst case, we have detected strong deviations that are caused by the fact that the function for Suffix Tree matching are not executed in atomic manner and may be preempted by timer interrupts and operating system activities as well as the dynamic properties of QEMU.

Considering the time required for generating and inserting a new node into the Suffix Tree, for best case it required at least 0.0402*ms* (= 12060 timer cycles) per node and 0.6199*ms* for average case. The median value was 0.3510*ms* that also shows a strong deviation from the average value. This happened due to the fact, that strong outliers have been observed in the worst case based on the same reasons as found for matching activity.

In fact, to provide precise and reliable values for worst case, appropriate code analysis methods have to be comprehensively applied for calculating worst case execution times of the Behavior Knowledge Base searching and construction as well as an evaluation on a fully deterministic platform.

9.4 Overhead of the Overall Approach

The overhead produced by the Online Anomaly Detection is composed of the overheads of the individual parts of the Anomaly Detection Framework:

1. the overhead for the system call extraction produced by the System Call monitor
2. the overhead for including the system call into the Behavior Knowledge Base's Suffix Tree
3. the runtime requirements for examining the system's health state by the OS Health Monitor
4. the execution time required to set the classification marker in the Behavior Knowledge Base

The overheads of the first three parts have been examined in detail in the experiments and have been discussed in the former sections. We did not explicitly measure the execution time for setting the classification marker. However, the time overhead produced by this final step is expected to be constant.

Summing up the worst-case overheads of the individual parts of the framework leads to the worst-case runtime costs of the overall approach. Nevertheless, this is a very pessimistic assumption, as here, the overhead concerning the OS Health Monitor includes the complete execution of the OS Health Monitor Center (in forms of a Workerthread) including all Component Monitor Modules. In practice, when applying anomaly detection to work online, the OS Health Monitor will only update the health (sub-)signals of those components that are effected by the executed system call. The OS Health Monitor will therefore processes only the Signal Generator rules that are associated with the system call which in fact is a small subset of the complete collection of rules processed by the Workerthread. By this, the execution time of the OS Health Monitor is reduced enormously. However, because each system call is assigned a different set of Signal Generator rules (different in number of rules and their complexity), it is not possible to formulate a general reference execution time value for examining the system's health state. This requires individual examinations of each system call. The execution time for examining the health state is a property of the system call (and is therefor constant) and has to be integrated into the worst-case schedulability analysis.

9.5 Summary

This chapter summarizes the evaluations conducted in terms of measuring runtime and memory overhead produced by the Online Anomaly Detection. Even though performed on different platforms as well as in the context of different scenarios, the presented results provide reference values for the overhead that has to be taken into account when applying the proposed approach. While the system call extraction consumes constant execution time independent of which system call is executed, the health state examination of a system call requires different (constant) execution times associated with the respective system call, but in any case bounded by the execution time of Workerthread executing the OS Health Monitor. Furthermore, the time required for adding the system call into the Behavior Knowledge Base depends on the position of the system call in the behavior sequence. Consequently, the runtime overhead generated by the Online Anomaly Detection depends, on the one hand, on the worst-case length of an application's system call sequence. On the other hand, it is determined by the specific worst-case structure of the sequence (maximum number of occurrences of particular system calls within a sequence). We can therefore state that the runtime evaluation of the Online Anomaly Detection approach can only be performed in an application-specific manner and has to be taken into account in the system's schedulability analysis.

Part V

Case Study

CHAPTER 10

Evaluation Methodologies

The development process of autonomous systems is different [49] as the key challenge is the implementation of the decision making part. Especially, the development becomes more challenging in terms of assuring dependability and reliability when being applied in the domain of intelligent mechatronic systems [50].

Evaluation of autonomous approaches such as the here proposed Online Anomaly Detection in not straightforward as compared to ordinary approaches because of the difference in the role of decision making processes: autonomous decisions instead of (programmed or) human-controlled [83]. Hence, methodologies for testing ordinary systems are not suitable here. A growing demand for research concerning testing and evaluation of autonomous systems can be observed. In this chapter, we discuss the problems and challenges and formulate requirements set on evaluating autonomous systems. Based on these requirements, we propose to use virtual environments for the purpose of evaluation and analyze their applicability with respect to the defined requirements.

10.1 Problems and Challenges

Several publications analyze the implicated problems and challenges of testing and evaluating autonomous systems such as Roske et al. in [83], Thompson in [88], Macias in [71] and Garrett in [48]. The latter, in particular, discusses to problem of evaluating Artificial Immune Systems as already introduced in Chapter 3.5. Pure static evaluation by mathematical methods or formal verification is not sufficient as the strength of such approaches lies in the effectiveness of the approach under operation. Garrett proposes to evaluate the usefulness by splitting the evaluation into *distinctiveness* and *effectiveness*. Distinctiveness (according to [48]) deals with the question whether an approach is distinctive in such a manner, that it cannot be transformed to other approaches applied for solving that particular problem. In fact, the validation of distinctiveness can be performed in form of static or formal analysis by showing the uniqueness of the approach (which we already have shown in [Stahl et al., 2013] and [Stahl and Rammig, 2014]). In contrary, effectiveness is concerned with showing that the evaluated approach results in better performance (in terms of obtaining better or faster results) compared to other methods, and therefore, it can only be measured at operation time within experiments.

Possible methods to evaluate such an approach in operation are model-based testing as proposed by [58]. However, models always provide abstractions that may unintentionally mask some system properties so that the according results will not be able to match the real system in operation.

Roske et al. specify the problem of testing and evaluation in more details in [83]. First of all, evaluation requires a *safe environment* because of lower tolerance in terms of unpredictable potential decision errors in real life applications. Similarly, Macias argues that a robust infrastructure is required to support measurements for testing and evaluation [71]. Secondly, the lifecycle of autonomous behavior of a system is composed of performing sensing the system and its environment, acquiring knowledge from the sensed data and building up a representation, analyzing the sensed data and executing the decision making. In this context, Roske et al. propose a clear separation of concerns for testing the individual lifecycle phases in order to ensure the correctness of each phase and to make the identification of a potential source of inadequate system performance possible:

1. **Testing the perception function**: Perception is concerned with observing and sensing the environment and its characteristics. The data delivered by perception defines the knowledge of *understanding* the system and its environment and usually deals as the basis for decision making. The correctness of perception function has to be tested in separate as erroneous perception may lead to fault decisions. Furthermore, it evaluates the control of hardware and environment including their failures and allows to perform a controlled failure injection.

2. **Testing the decision making function**: The decision making process is implemented according to a specified objective function. The evaluation of the decision process mainly addresses to verify its correctness by considering whether the performance of the system caused by autonomous decisions approximates towards the objective function. The decision making function has to be tested in separate as it may evolve due to being applied in complex adaptive systems.

3. **Testing the execution function**: Testing the execution function is concerned with the testing ability to execute the decisions in order to obtain confidence about the system performance in terms of classical system functions (control performance, protection, reliability, etc.) as well as physical performance (speed, capacity, resource demand, etc.).

Thompson in [88], emphasizes that evaluating the effectiveness of autonomous systems shall be explicitly concerned with testing against a specified set of requirements dictated by test missions set on the system performance (e.g. aspects of objective function) instead of testing the decision process based on assumptions made on decisions. Therefore, a definition of test scenarios is essential, combined with scenario parameters and metrics that quantify the satisfaction of the specified requirements dictated by the test mission. According to Macias [71], experiments must be repeatable, controlled, and reproducible and require instruments to provide insight into the system functioning as well as to track and record its status. Furthermore, Macias points out that the evaluation environment must be as adaptive as the autonomous approach as, from his point of view, the evaluation of the autonomous approach shall be performed in parallel to its development. Any modification of the autonomous approach requires an adaptation of the evaluation concept, of the experiments and probably of the environment as well.

Thompson [88] suggests to use virtual environments to test autonomous software. This conforms with the proposals of Roske et al. [83] and Macias [71], as virtual environments allow a system to be preliminarily executed in a safe environment preventing damage or harm of the target system and its environment. Furthermore, virtual environments are flexible and adaptable to changing requirements. Thompson claims that (see [88]) "virtual testing must become a standard complement to field-testing UASs" (UAS: abbreviation for *Unmanned Autonomous Systems* used by Thompson) "..if the testing community is ever going to be able to test an intelligent UAS safely and comprehensively". Furthermore, he argues that the most important requirement is that the virtual environment is identical with the real application environment in such a way that all essential physical properties and obstacles match the real-world ones (e.g. in terms of dimensions and resolution). Additionally, the virtual environment must provide all necessary environment information and requires to be interaction-based in order to enable the target system to take all actions according to its specification. To ensure comprehensive evaluation results, it must be able to extract precise and suitable operation data from the virtual environment. Thompson's specifications of requirements on virtual environments for testing autonomous systems are superimposable with the properties Macias describes.

There is no state-of-the art methodology to realize an evaluation for autonomous approaches up to now and developing a sophisticated test methodology is not the scope of this thesis. However, the problems and challenges discussed in the research community mainly guided our idea in order to perform a systematic and structured evaluation of the Online Anomaly Detection for which we first specified requirements (which we also published in [Stahl et al., 2015]).

10.2 Requirements

The basis for specifying requirements is a clear definition of the objective of the evaluation. Referring to the assumption that the evaluation of autonomous approaches in terms of effectiveness has to take place when the system is in operation, the evaluation is usually application-specific and can only be conducted within its application domain.

The mission of the evaluation of our Online Anomaly Detection is to verify whether the anomaly detection approach works correctly, according to the properties that have been intended in the implementation. As a basis, the Anomaly Detection Module has built up a Behavior Knowledge Base in order to establish the system's normal behavior based on the behavior of the applications tasks (defined by system calls invoked by the tasks). Changes in the applications, their parameters and performance, changes in the environment as well as in the system itself may lead to system reconfiguration and, respectively, result in changes in the system behavior. Detecting these behavioral changes is the crucial task of the Online Anomaly Detection. All occurring behaviors have to be classified by the Online Anomaly Detection. The classification method analyzes the behavior with the objective to declare observed behavioral patterns either to be normal, suspicious or dangerous by considering system health signals provided by the OS Health Monitor. One of the main objectives is to verify whether the Online Anomaly Detection classification method works correctly: This means that behaviors shall be classified as *normal* if the system is expected to be in a healthy state. In contrary, in case of a instability of the system or even a failure, it is the question whether this unhealthy state will effect the classification outcome of the Online Anomaly Detection. To realize this, means to induce and control changes in the executing system itself and the environment are required, also including unsafe or dangerous system states, of course, initiated in a controlled manner.

The detection quality of the Online Anomaly Detection has to be examined in order to draw conclusions on the reliability of the detection mechanism of the approach.

Based on this objectives, we set the following requirements on the evaluation environment and its architecture:

R.1 **Reconfigurable applications and environment**: Our approach is designed to be integrated in a real-time operating system that serves self-reconfiguring applications. It is designed to detect unintended and malicious system states provoked by autonomous decisions at the application or operating system side. Therefore, as a basis to assess the effectiveness of our Online Anomaly Detection, an application environment is required that is dynamically changing with a real application implementing autonomous behavior.

R.2 **Execution platform**: Referring to the previous requirement, an execution platform for the real-time operating system is required that is suitable for the application environment. In fact, as the Online Anomaly Detection is integrated in the real-time operating system ORCOS, we need an execution platform that is able to run ORCOS and satisfies the requirements of the application purpose.

R.3 **Safe environment**: In accordance with the challenges described above, a safe environment is required for evaluating the Online Anomaly Detection. Safe environments stem potentially resulting threats caused by the uncertainty of autonomous behavior. Thereby, the environment has to be identical to the real one (or at least as close as possible to the real one concerning the relevant system properties) and, on the other hand, it has to enable full control over the system entities. The latter is related to e.g. controlled initiation of dynamical changes, also including injection of failures in system components, that lead to changes in application or system behavior or even to reconfigurations. Changes initiated in a controlled manner build up the basis for assessing performance of the Online Anomaly Detection in terms of detecting these changes and classifying the resulting behaviors.

R.4 **Separation of evaluation concerns**: Based on the requirements formulated by Roske et al.[83], evaluation environments shall make sure that clear system boundaries can be guaranteed. This addresses the system boundaries between perception function, decision making function and execution function. The individual system parts have to be isolated in order to ensure a decoupled evaluation. From the viewpoint of the Online Anomaly Detection, the perception function is composed of two entities: the application behavior represented by system calls, and the input health signals delivered from the OS Health Monitor. The health signals, in turn, are determined by sensing the health parameters that reflect the internal states of the operating system components.

R.5 **Interaction-based environment**: Dynamical changes in the environment or the system itself are the foundations of autonomous reactions. The evaluation environment is required to provide instruments in order to induce them in a controlled manner as already emphasized within the requirement concerning a safe environment. Basically, this can be realized by allowing interaction between the executing system and the operator responsible for the evaluation.

R.6 **Evaluation output**: To address the challenge for the experiments to become repeatable, controlled, and reproducible, the evaluation environment must integrate instruments to monitor and record the execution trace of the evaluation scenario. Of course, we are mainly interested in the performance of the Online Anomaly Detection with respect to the question of whether the intended properties that have been implemented can be achieved in the execution. Hence, an essential requirement is to obtain information that reflect the output of the evaluation. On the one hand, it is the operating system ORCOS that has to deliver evaluation results as the target evaluation approach is integrated into it. Furthermore, behavior of the application and the environment and their changes must also be traced in such a manner that it is possible to set this data in context to the internal behavior of the Online Anomaly Detection. The evaluation environment has to ensure full traceability of the actions performed by the contributing system components of the evaluation scenario. By this, we will be able to draw conclusions on the anomaly detection performance from the according evaluation scenarios.

The applied evaluation environment must satisfy these requirements in order to allow a comprehensive evaluation of the Online Anomaly Detection. As claimed by Thompson [88], Virtual Reality and Virtual Environments are providing adequate characteristics for evaluating autonomous systems. Therefore, we analyze their applicability for evaluating the Online Anomaly Detection, especially, in terms of meeting the specified requirements.

10.3 Applicability of Virtual Reality and Virtual Environments

The former sections have already indicated the applicability of Virtual Reality and Virtual Environments for evaluating purposes. Before discussing the potentials of these approaches in terms of evaluation and in particular in terms of evaluating the Online Anomaly Detection, we will first have to clarify some common conceptualities as well as tointroduce some basic technology concepts.

Virtual Reality

Virtual Reality (VR) is defined as a technology that "...replicates an environment that simulates physical presence in places in the real world or imagined worlds" [18]. It enables the user to be put into a virtual three-dimensional world that is a computer-based replication of a real counterpart. Today, VR technologies, initially applied in the computer gaming field, are used more and more for design reviews, in the field of mechanical engineering, and plant engineering, and thereby support the planning, the design and development phase as well as the evaluation of technical systems [86].

Virtual Prototype

A Virtual Prototype (VP) is a computer-internal representation of a real product or a real prototype [86]. It is constructed based on information on the (physical) shape and the structure of the product (e.g. 3D-CAD model), its kinematics and dynamics, and its specification for information processing. A VP is considered as the virtual counterpart of a real-world object.

Virtual Environment

[86] defines a Virtual Environment (VE) as "... a synthetic computer-generated environment, which presents a visual, haptic, and auditive simulation of a real world to a user..". It is composed of (multiple) VPs that build up the virtual counterparts of the environment. A VE that is applied for product development, analysis and evaluation requires the integration of the product's VP.

VR enables to test mechatronic systems, even in early development phases. Especially, the virtual prototype inside a virtual environment provides suitable instruments as it replicates the real system in its environment. For using virtual environments to evaluate autonomous approaches, the requirements imposed for the evaluation environment (namely **R.3** to **R.6**) have to be fulfilled.

VE and VP, by concept, are constructed based on real environments, real prototypes or devices (respective to e.g. the size, mass, physical behavior and so on). Hence, VE and the VP that operate inside act like their real counterparts. Using a virtual prototype inside a virtual environment provides a tool to safely verify all aspects of the mechatronic system without endangering neither the real hardware nor the real environment. Therefore the requirement **R.3** of providing a safe environment is fulfilled by the nature of VR. Every essential system entity of the real system is constructed independently as an encapsulated system entity in its virtual counterpart represented by software implementation. This enables full control over the properties and attributes also in terms of inducing changes which is a fundamental requirement for evaluation. Furthermore, the encapsulated virtual counterparts of the system entities replicate the offered interfaces which enables components to be exchanged by another (e.g. simpler) implementation. By the encapsulation of the system entities, decoupled evaluations of individual system parts become possible. This, in particular, forms the foundations for realizing the separation of concerns in the evaluation (see **R.4**).

VE can provide interfaces for access to system entities and their parameters for analysis, assessment as well as manipulation purposes. Using these interfaces at run-time for changing system components, modifying the virtual environment or even injecting failures into the system allows controling the system in an interactive manner according to requirement **R.5**.

VE offers visual representation of the system under execution within its environment and, thereby, forms one kind of evaluation output as it makes the system performance directly observable (in contrast to model-based or simulation-based evaluations). Furthermore, instruments for visualizations of parameters, including values of physical components as well as parameters of the executed software provide facilities to visualize all parameters needed for the evaluation, and make evaluations traceable as defined by requirement **R.6**.

With all these characteristics described above, the concept of virtual environments meets all requirements which are imposed on it for evaluation purposes. The powerfulness and flexibility offered by virtual environments makes it a vital instrument and suitable for the evaluation of the presented Online Anomaly Detection for self-x real-time systems.

10.4 Summary

This chapter addresses the problem of determining an evaluation technology for the Online Anomaly Detection. This is not straightforward, as no state-of-the art methods exist up to now. Evaluating autonomous approaches requires new methodologies to address the problems and challenges that have been identified by several researchers dealing with that topic. Furthermore, these new methodologies must guarantee full traceability of evaluation experiments even though the system's behavior is fully dynamic.

We present the objectives of the evaluation of our Online Anomaly Detection and formulate requirements set on an evaluation environment that, of course, implies the presented problems and challenges. We discuss the potentials offered by VR technology and prove, based on the requirements, the applicability of virtual environments for evaluation purposes. By this discussion, we establish the basis for applying the Virtual Evaluation Environment for the evaluation of the Online Anomaly Detection as presented in the next chapter.

CHAPTER 11

Evaluation Case Environment

Virtual reality technology as introduced in the former chapter offers an opportunity to manage the execution of autonomous systems and provides visualization techniques that are used to present the system behavior. Because of its great potentials to realize the requirements set on the evaluation (see previous chapter), we have decided to use a virtual environment to evaluate the effectiveness of the proposed Online Anomaly Detection approach. Applying virtual reality makes the evaluation process very comfortable and the results (immediately visible) intuitively conceivable.

In this chapter, we present the application context chosen for the evaluation of the Online Anomaly Detection and the respective Virtual Evaluation Environment designed for it. Furthermore, we introduce the application-specific implementation and extensions in the operating system and the Online Anomaly Detection that had to be integrated because of the chosen application context. We finalize this chapter by specifying evaluation scenarios that are dedicated to ensure a comprehensive case study for the anomaly detection approach proposed in this thesis.

11.1 Evaluation Environment

The Online Anomaly Detection integrated into ORCOS is executed in order to evaluate a task's behavior with respect to its effects on the system's health state. The evaluation of the Online Anomaly Detection requires a real application context for ORCOS (**R.1**) as well as an execution platform (**R.2**). In some previous work, ORCOS was ported to be executed on a miniature robot BeBot[1] that was designed for autonomous driving applications in dynamic environments. Hence, using the BeBot as the application context lead to the evaluation environment applied here.

As a foundation, we have exploited an evaluation environment called *virtual test environment* that was priorly built up for the purpose of validating the code coverage of self-optimizing user tasks executed on ORCOS [86]. The evaluation environment mainly consists of a Virtual Environment building up a landscape for the miniature robot BeBot[1] in form of a labyrinth/maze (see Fig 11.1). The BeBot is aimed to autonomously drive through a maze in order to find a designated destination. The BeBot's driving- and destination-finding strategy is implemented as an application task for ORCOS. Several strategies are implemented so that it is possible to exchange the executing strategy by means of system reconfiguration.

For evaluating the Online Anomaly Detection, the virtual test environment as designed for the work in [86] had to be adapted and extended, and resulted into the here presented evaluation environment. The concept of this evaluation environment was published at the HCI International 2015 in [Stahl et al., 2015].

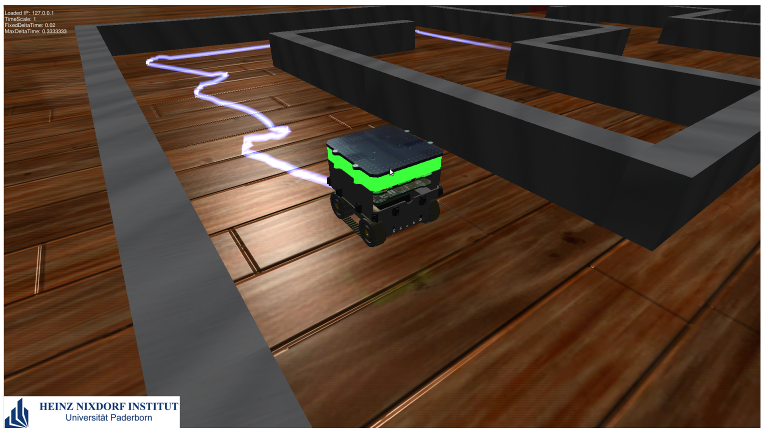


Figure 11.1: Virtual Evaluation Environment for the Online Anomaly Detection.

11.1.1 Virtual Evaluation Environment

In order to reduce the implementation effort, for evaluating the approach presented in [86], a third party solution Unity 3D [91] was selected. Unity 3D is an IDE for developing 3D applications mostly used in the area of computer games. It provides the developer with many comfortable solutions for building up virtual environments. Furthermore, it allows to include own code by a powerful scripting interface for C# and thereby enables to integrate individual algorithms and complete implementations inside Unity 3D. It is executed on a general purpose operating system (here Linux). As in [86], the virtual test environment was built up on top of the VR system Unity 3D and integrated the BeBot with ORCOS running on it, we exploit the implementations done as a basis to implement the Virtual Evaluation Environment for the Online Anomaly Detection.

The Virtual Evaluation Environment consists of a VP for the miniature robot BeBot (introduced in detail in Section 11.2), and its VE being a randomly generated maze. The VE and the VP have been constructed based on the (physical) properties of the real prototype and the real environment in which it operates. The maze consists of rectangularly arranged walls and is built up with virtual counterparts of brick-like plastic elements used for testing the real BeBot. The look of the environment way is polished by adding some graphical effects just for a better look, but nevertheless, it replicates the real environment as required by **R.3**.

As the task of the robot is to find a way through the maze to a dedicated destination, the algorithm for finding the destination in the maze is implemented as a task of the underlaying real-time operating system ORCOS executed on the VP. The start position of the BeBot and destination position are defined (randomly) inside the maze. Thereby, the randomly generated maze acts as a test environment for the implementation of the autonomous algorithm.

In order to ensure the separation of the evaluation concerns (**R.4**), in our Virtual Evaluation Environment, all system parts such as virtual (hardware) components, and the executed RTOS with its tasks communicate with each other by dedicated interfaces that correspond to the interfaces of the real counterparts. This architecture enables the exchange of system entities and hence, decoupled evaluations of individual system parts, in particular, the environment, the components of the VP, the execution part composed of RTOS and its tasks and fulfills **R.4**.

R.5 requires changes in system components, modifications of the virtual environment and failure injection to be performed in an interactive manner. In our concept, we enable the user of the system to implement such interactive behaviors very easily. A scripting language is provided for the users to build their own interaction methods quickly. Furthermore, the Virtual Evaluation Environment was extended by a GUI (Graphical User Interface) for manipulating the properties of the VE and the VP (see Section 11.3).

Our Virtual Evaluation Environment offers a visual representation of the scenario execution. Visualizations of diverse parameters, including values of physical components as well as parameters of the executed software as required for evaluation in **R.6** are provided by the implementation extensions made on the Virtual Evaluation Environment. One means, e.g., is the notification of the commands that the BeBot executes.

The resulting Virtual Evaluation Environment ensures that the system and its behavior can be analyzed inside its virtual world with a close interaction with the virtual objects.

11.1.2 Architecture

To realize this Virtual Evaluation Environment, an architecture is required that combines the VE with its VP and the operating system ORCOS as the execution platform for the BeBot application.

Figure 11.2 shows the architecture we developed for realizing this. It contains the VE which generates the environment (maze) and the VP (BeBot). For executing applications (strategies for finding destination) on BeBot, we use ORCOS. However, ORCOS is not able to be directly executed on the BeBot within the VE. In contrary, ORCOS runs on the emulator *QEMU* (see section below) that is connected to the VP over an *Assistant Communication Module* (ACM, see section below). Once the virtual test environment is executing, ORCOS starts to communicate with the VP through ACM. The environment data provided by VE is transferred back to ORCOS. A particular BeBot controlling task processes these environment data and commands the virtual BeBot by sending messages to it. This entire architecture of the virtual test environment is implemented on a Linux system.

QEMU

QEMU (Quick Emulator) is a generic and open source machine emulator and virtualizer [4]. QEMU is a very fast virtual machine because it applies dynamic translation technology to emulate the processor so that it can achieve good emulation speed. As a virtual machine, it allows the users to run different operating systems on a specific emulated virtual hardware platform. Unlike other popular virtual machines, QEMU has two operating modes to run the operating system. The first one is to emulate a full system like a full PC including the processor and other peripherals. It can be used for general purpose operating systems like Windows, Linux or Unix, and even Mac OS. The other mode called *user mode* is to emulate a user specific hardware platform. This mode can be exploited to launch a process compiled for one CPU on another CPU [4]. In our case, ORCOS is able to be executed on an emulated PowerPC 405 CPU under x86 architecture, even though ORCOS does not support that architecture. In our virtual environment architecture, the user mode of QEMU is applied.

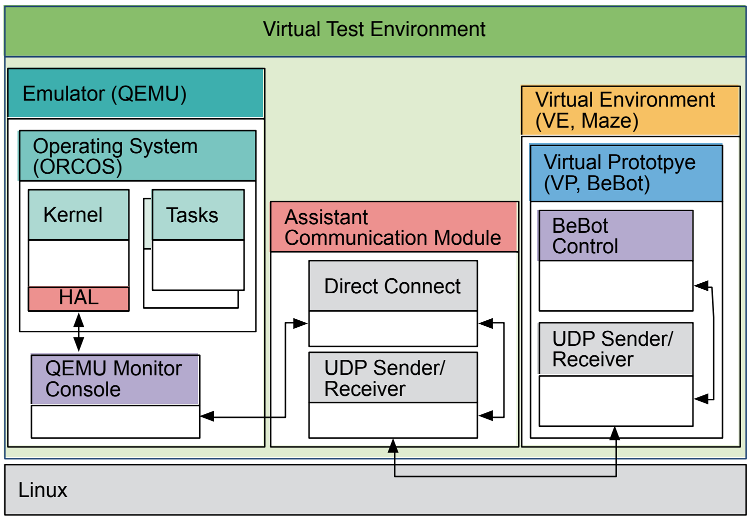


Figure 11.2: Virtual Evaluation Environment Architecture.

We use QEMU version 1.5.0 because this version enables us to realize the communication with the virtual environment over the network. QEMU is still under development, but it has dropped further development of the PowerPC 405 processor family which, in fact, is the main development hardware of ORCOS. As QEMU has not been specifically designed for ORCOS, it leads to some issues of compatibility. Unfortunately, the QEMU version used in our environment only supports a fixed interval timer instead of an original programmable timer of the PowerPC 405 as required by ORCOS. Moreover, since dropping the development of PowerPC 405 emulation, both, PCI and integrated Ethernet card in CPU, offer only empty wrappers that require to be implemented. Consequently, it is impossible to implement the network communication through adding the missing hardware drivers if using the official QEMU version. So, based on this, a customization of QEMU is required.

The fixed interval timer in QEMU version 1.5.0 overwrites the instance of the programmable timer during the initialization of CPU. This problem has been solved by disabling the fixed interval timer in the source code which makes the programmable timer implementation possible to be executed and makes the timing measures more reliable.

On the other hand, the communication problem is a little tricky. Integrating a network card chip into the emulated PowerPC 405 CPU is provided as an option in the setup of QEMU. However, QEMU simplifies the implementation of emulating PowerPC 405 with abandoning this network card chip, which makes the communication between ORCOS and the outside world impossible by using this option. Furthermore, QEMU is not offering direct access to hardware devices installed on the host system so that using the network device of the host is also not possible.

In order to realize communication between ORCOS through QEMU and the VP in our VE despite the problems explained above, we implemented a workaround. A monitor console feature of QEMU provides a way for interacting between QEMU and an external software. Through specific commands, the monitor allows to inspect the running guest OS, including changing removable media, taking screenshots, monitoring the memory content, and controlling other aspects of the virtual machine [81]. QEMU monitor console can be accessed over TCP socket connection. With this, through sending a specific query to QEMU an internal function is triggered and the monitor console is able to receive the corresponding result. Such a specific query must follow the *QEMU Machine Protocol* (QMP) allowing applications to control a QEMU instance. QMP is a lightweight, text-based protocol, which makes it easy to parse data format and allows to add new monitor functions due to the scalability of QMP.

Exploiting the properties of the monitor console enables to implement an unidirectional communication between the monitor console and an external software. For our purpose, communication has to be bidirectional. In order to enable an external software to send the messages to QEMU, the monitor console must have authority to write into the QEMU internal memory space. Unfortunately, there is no such feature offered by the QEMU monitor console. Hence, we enhanced QMP with adding a new query function to the QEMU monitor so that we could modify the memory content through sending the correct monitor query.

In order to realize the VE to communicate with QEMU, we have implemented an external software called *Assistant Communication Module* which builds a part of our evaluation environment.

Assistance Communication Module

Assistant Communication Module (ACM) is for achieving the communication between QEMU and the VE. The process of communication is specified in Figure 11.3. ACM is separated into two parts: One integrated as a driver in ORCOS (written in C++), and its counterparts as an independent executable software (written in Java). These two parts are interconnected and exchange data by using the QEMU monitor console. The ACM part in ORCOS synchronizes messages with the other part of ACM through memory buffers. The receiver buffer as well as the sender buffer are implemented as static memory ranges, the receiver for saving the received message from while the sender for saving the outgoing messages to the monitor console. As mentioned in the previous section, the QEMU monitor console is able to aid the external part of the ACM to view and modify the memory content. Therefore, the external ACM checks modifications of the receiver and the sender buffer with high frequency in order to guarantee the synchronization of the buffers. The protocol design of the receiver buffer and sender buffer is inspired by TCP/UDP protocol.

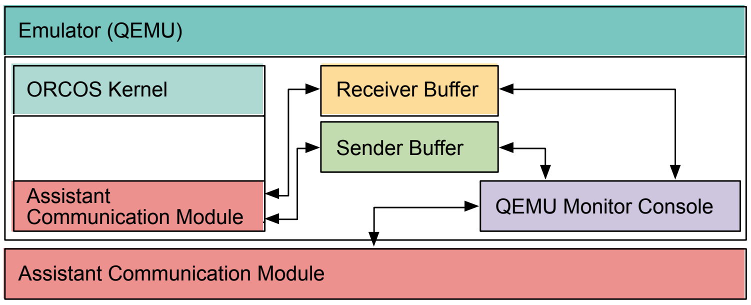


Figure 11.3: Architecture of the Assistance Communication Module.

11.2 The BeBot

The BeBot is a small robot serving as a technology platform used for research in the domains of dynamic reconfigurable systems. Its chassis is shown in Fig. 11.4. ORCOS was ported to execute on the BeBot in order to implement dynamic and reconfigurable applications for controlling the BeBot.

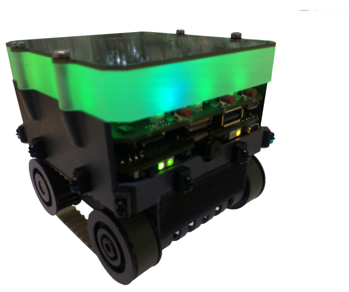


Figure 11.4: The BeBot mini robot.

11.2.1 Technical Specification

The mini robot has a dimension of ca. $9cm\ x\ 9cm\ x\ 7cm$. It provides 12 infrared (IR) sensors for sensing its environment. The BeBot is equipped with microcontrollers to process the data of the IR sensors and an I^2C bus to transfer the IR sensor data. The robot movement is realized by two 2W DC gear motors which actuate two chains, one for each side of the BeBot. The BeBot comes with an ARM7-based processor with 520 MHz and 64 MB memory. The detailed technical specification of the BeBot as well as a programming guide can be obtained from [1].

11.2.2 Virtual Prototype

The BeBot is one main object in the Virtual Evaluation Environment as the execution platform for ORCOS and its applications. Therefore, the BeBot was implemented inside Unity 3D as a virtual prototype in order to represent a virtual counterpart of the physical device. The shape model is based on a CAD model so that its appearance is nearly as realistic as the real BeBot. The existing 12 IR sensors to detect distances to obstacles, the gear motors and the chain drive have been realized as components in the VP that implement their (physical) functionalities in order to replicate the behavior of the physical BeBot as close as possible to its real behavior. The Virtual Prototype implements the same low-level interface to the underlying hardware as the real BeBot. Commands that control the BeBot have been integrated into the VP in order to control the BeBot's virtual counterpart in the same manner. By this, for the applications, it becomes transparent whether the application is executed on the real hardware device or on the VP.

The BeBot application (described below) mainly operates on the values the IR sensors deliver. To guarantee a realistic simulation of the physical properties of the IR sensors, the real IR sensors have been examined concerning the distance values they deliver for dedicated distances to obstacles. The IR sensors installed on the BeBot are physical rays that sense the environment and deliver `int16` values returning the distance to the obstacle it intersects. Fig. 11.5 shows a model of the BeBot with its 12 IR sensors and their according physical rays.

To produce identical values as will be received from the real IR sensors, first, we measured the real behavior by a number of test series. The values of the real sensor have been read out at different distances to an obstacle. We averaged the measured values and used this as a reference for the virtual prototype, as seen in Figure 11.6. In the test series, we measured the value per every 1 cm. At a distance of 11 cm, no reliable results are received, which leads to the assumption, that the sensor is only able to detect obstacles up to 10 cm. For our simulation, these reference values r are stored in an array A. For distance d received from the physics engine $d_{floor} = \lfloor d \rfloor$ and $d_{ceil} = \lceil d \rceil$ are calculated as the two adjacent indices for the array. By

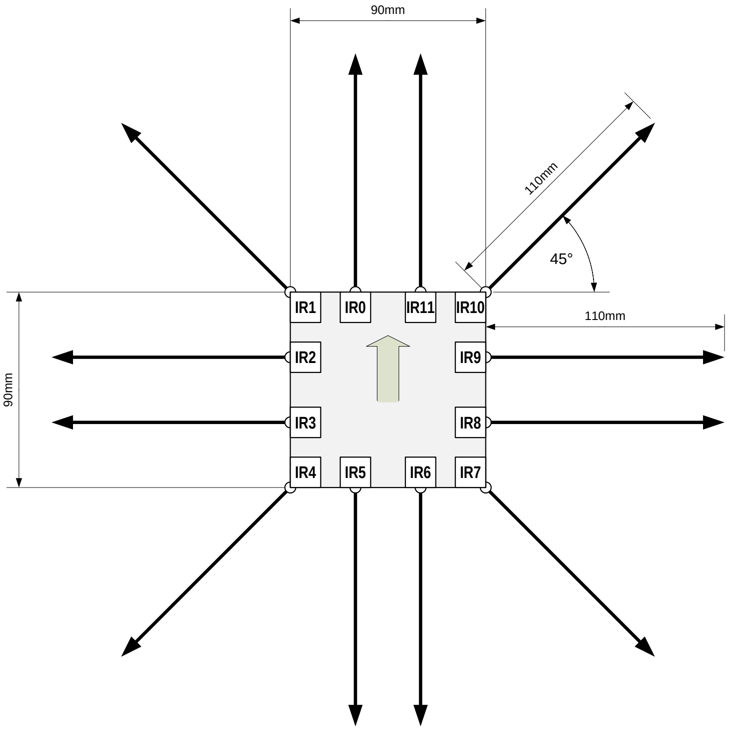


Figure 11.5: Model of BeBot with geometric dimensions, its IR sensors and sensor ranges

reading out the corresponding values of the array, we get $r_{floor} = A[d_{floor}]$ and $r_{ceil} = A[d_{ceil}]$. Between these two values we linear interpolate using the fraction part $\{d\}$ to calculate the virtual sensor value $r = r_{floor} + (r_{ceil} - r_{floor}) \cdot \{d\}$. For reflecting the reality, we added approx. 10% random noise to the virtual sensor value as we found out that this is the amount of jitter we received from the sensors.

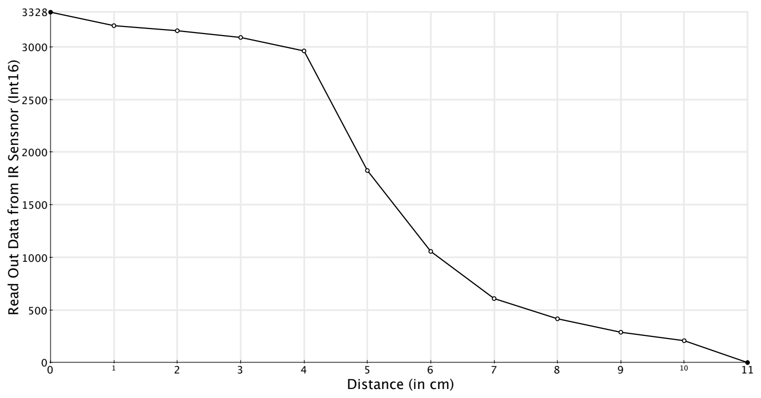


Figure 11.6: Mapping of IR Sensor values measured for dedicated distances (in *cm*).

In addition to the accurate physical model of the real IR sensor behavior, we provide alternative implementations that have been useful for our evaluation. The following provides an overview of all implementations of the IR sensor.

1. **AccurateModel**: the accurate physical model obtained by measuring the real IR sensors with noise factor

2. **IdealizedModel**: the model obtained by measuring the real IR sensors without the noise factor
3. **LinearModel**: the IR sensor values are linear to the distance
4. **ExtendedRangeModel**: extension of the range of the physical ray up to 22 *cm* in order to enable sensing greater distances

The user is allowed to select one of the implemented models of IR sensor behavior according to the requirements of the evaluation scenario. In order to realize failure injection, we have extended all these models by following failure models:

FailureModel1 IR sensor delivers value 0 (no obstacle detected)

FailureModel2 IR sensor delivers *max_value* (obstacle close to sensor)

FailureModel3 IR sensor delivers a fixed value set by the user

FailureModel4 IR sensor delivers a random value

The failure injection implementation was extended by introducing a latency into the reading out procedure of the sensor. This was done based on the assumption that each IR sensor is equipped with a simple microcontroller that in case of a failure (e.g. a defective contact or a memory fault) requires more time to return the distance value. This, basically, effects the communication time between the VE and ORCOS via ACM when requesting the according faulty sensor value. (The reason for this extension will become more clear in the description of the evaluation scenarios as well as in the discussion of the evaluation results.)

The implementation of the virtual IR sensors is not limited to the characteristics described above. Furthermore, the VP implementation offers an interface to integrate further models for IR sensor behavior as well as for the sensor failure.

Besides the IR sensors, the BeBot gear motors have also been implemented according to their physical specification. Specified commands trigger gear motors to drive the BeBot through its environment. For evaluation purposes, we included a method which manipulates one or both gear motors and allows to decide whether a motor will not react at all, randomly react correctly on a driving command or just react randomly whatever value is sent.

The implementation of the VP fully replicates the behavior of the physical device and makes the BeBot's virtual counterpart becoming as realistic as possible. By this, the BeBot becomes a suitable execution platform for the desired evaluation.

The experiments described in this evaluation have been conducted within the Virtual Evaluation Environment only. Therefore, we will use the name BeBot as a synonymous for the virtual prototype as the physical device is not of interest for this evaluation.

11.2.3 The BeBot Application

The goal of the BeBot in the virtual environment is to reach the destination in the maze. Basically the BeBot has no knowledge about the structure of the maze. Furthoremore, the BeBot is just a robot vehicle without any intelligence. The intelligence lies inside the application task that only operates based on the values of the IR sensors. Hence, we designed several maze-solving algorithms to control the BeBot in order to reach the destination more or less efficiently:

Right-side wall follower: This algorithm is simple and best-known for solving path finding in mazes in which all walls are connected together. The robot always drives forward by following the right-side wall of the maze. Thereby, the robot must be guaranteed not to lose the wall it is following until it reaches the destination of the maze. For mazes that are not simply connected, this algorithm can not guarantee the goal to be reached.

Left-side wall follower: This algorithm follows the same principles as right-side wall follower. The only difference is that the robot follows the left-side wall of the maze instead of the right-side wall.

Random mouse algorithm: In this algorithm, the robot makes random decisions about the next direction to follow whenever the robot meets a junction. For robot path-searching, this algorithm does not guarantee that the robot will find the right solution. However, in contrast to both formerly described algorithms, it can be applied in mazes in which not all the walls are connected (e.g. loops) and will probably find its destination.

Parameterized random mouse algorithm: As an improvement of random mouse algorithm, this algorithm makes a parameterized-random turning decision in which the weighted parameters are based on the distance calculated by the position data between the robot and the destination. This algorithm required an extension of the BeBot by a GPS-like position module. For executing this algorithm, the BeBot gets information about the destination's position and its own and by using this information, it can calculate its linear distance. Applying this algorithm, we expect an enhancement of the probability of the robot to find its destination.

Each of the maze-solving algorithms is best applicable for a particular environment scenario. Figure 11.7 specifies the application side architecture that implements the decision making part to determine the most suitable strategy for solving the maze. Each strategy is implemented as the single task and stored in the task repository. The *Task Controller* is a simple analyzer that determines the effectiveness of the current searching strategy according to its objective function. For instance, an applied strategy could be assessed to be not suitable if the BeBot passes the same position over multiple times. In the case that the applied strategy is not suitable to the underlying maze, the *Task Controller* has to choose a better algorithm.

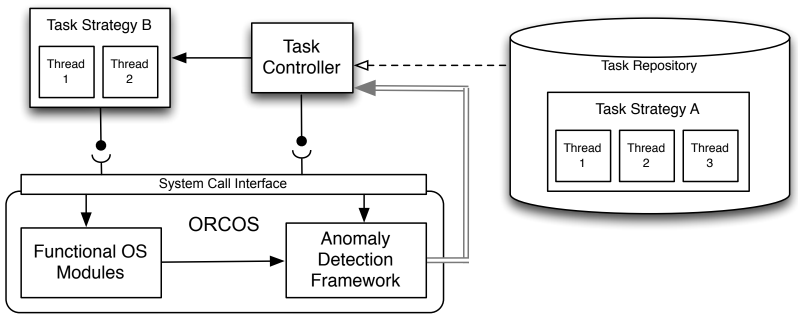


Figure 11.7: Application evaluation architecture.

The algorithm is implemented by using the ORCOS system calls to accomplish its job. Behaviors of the functional OS modules are recorded by the Online Anomaly Detection, which also analyzes these behaviors to determine whether they are normal or anomalous. If the

Online Anomaly Detection recognizes any anomaly associated with system calls made by an application, e.g. an unreliable device, such as a not correctly functioning IR sensor, on which access (through system calls) the algorithm relies on, the Anomaly Detection Framework supplies the *Task Controller* with these information. In such a case, the *Task Controller* will classify the algorithm to be not suitable anymore and has to choose another algorithm strategy from the task repository in order to meet the detected novel circumstances.

In the concept of the self-x architecture, the controller is part of the operating system. Thereby, it gains only the information relevant for the operating system. The *Task Controller*, however, requires more knowledge about the application context by analyzing the application's performance with respect to its objective function (e.g. detecting positions that have been reached multiple times). This knowledge, by the concept of the self-x architecture, is not available in the operating system internals. Therefore, in our application scenario, we have located the *Task Controller* into the user space in order to implement this knowledge.

11.2.4 BeBot Device Driver

For the applications, ORCOS provides system calls as interfaces to control hardware devices. The system calls offered by ORCOS for controlling the BeBot are enumerated in Appendix A.10. For serving the BeBot-related system calls, the BeBot Device Driver was implemented and is responsible for encoding the BeBot commands into a format that the BeBot can execute. BeBot Device Driver is an implementation of the abstract Device Driver Interface of ORCOS. However, the BeBot is controlled by commands over a local I^2C bus. Therefore, the BeBot Device Driver that implements the system calls is responsible to construct and encapsulate the I^2C bus commands. Moreover, as we do not operate on the real BeBot, the I^2C bus commands cannot be directly sent over the I^2C bus. The communication with the BeBot is masked. This is realized in a lower layer, the HAL, in order to make the communication between the application and the BeBot transparent for the device driver. The I^2C commands are sent to a specific address. Instead of using the I^2C bus, the according I^2C commands are encoded into UDP/IP packages which ORCOS sends via a network connection to the VE and the VP, respectively.

11.2.5 IR Sensor Monitor

As one of its functionalities, the BeBot Device Driver has been implemented to get access to the IR sensor values. The BeBot does not provide any self-diagnostic feature to announce sensor failures so that it is the task of the IR Sensor Monitor to detect whether all sensors are still reliable. To be able to monitor the health state of the BeBots IR sensors we have implemented an IR Sensor Monitor as an example to illustrate how expert knowledge can be integrated into the OS Health Monitor. The IR Sensor Monitor builds up one of the Component Monitor Modules of the OS Health Monitor. It collects the data from the BeBot 's IR sensors in order to check whether they are working correctly. The IR Sensor Monitor is responsible for evaluating the health state of all the 12 sensors on the BeBot, as usually a suspicion cannot be detected by analyzing one sensor in isolation but it is furthermore identified in the context of the values of the neighboring sensors. The IR Sensor Monitor also stores history sensor data. The health parameters recorded by the IR Sensor Monitor are listed in Appendix C.7.

11.2.6 IR Sensor Monitor Signal Generator

Failures in the IR sensors have to be detected by the rules defined for the Signal Generator of the IR Sensor Monitor. Four failure models have been introduced to the IR sensor. However, specifying Signal Generator rules is not straightforward as argued by the illustrated problem:

First of all, the values delivered in case of a failure (independent of the applied failure model) also belong to the range of legitimate values. For example, a distance value of 0 is returned if no obstacle is detected within the range of the sensor, but if the sensor is defective (applying **FailureModel1**) it will also return the value 0 (analogous argumentations can be formulated for all the specified failure model from **FailureModel2** to **FailureModel4**). Hence, we cannot definitely testify the status of an IR sensor based on only considering the value that the sensor delivers.

The IR Sensor Monitor stores the history data of the previously measured distances. Also, only considering the history values of a sensor is not sufficient to definitely identify a sensor failure: in case of a failure, for example, the value 0 (again if applying **FailureModel1**) will be delivered at any time the sensor value will be requested and therefore will be stored in the history. A multiple occurrence of the value 0 recorded in the history data, however, can also occur if the sensor does not sense any obstacle for a longer period of time. From this follows that a sensor failure cannot be detected by only considering the individual values of an IR sensor.

A step towards evaluating the health state of an IR sensor is that the sensor value has to be set in context with values delivered by other - in particular, its neighboring - IR sensors. For example, if one IR sensor delivers the value 0 and all its neighboring IR sensors also deliver 0 value, then, in all probability, there is no obstacle detected on this side of the BeBot. In contrary, if one sensor delivers the value 0 but all its neighboring IR sensors deliver values different from 0 (they sense obstacles in the environment), the situation is different. On the one hand, the probability is high that the 0 value delivering IR sensor is defective. On the other hand, such a situation can occur (even if seldomly) as illustrated by the Fig. 11.8. In this example, $IR8$ delivers the value 0 because, in contrast to its neighboring IR sensors, the wall is out of the sensors range. Again, considering only the IR sensors values of neighboring sensors does not lead to a clear outcome for the state of an IR sensor. However, if the history of the values is taken into account, the results becomes more reliable: referring to the previous example of one IR sensor delivering value 0 while its neighbors deliver values different from 0, the situation is a rather rare combination. Hence assuming that the BeBot moves in the meantime, the IR sensor values, especially of the IR sensor that previously delivered 0, shall change. If that one IR sensor value remains 0, the probability is high that this IR sensor is truly defective.

Based on this problems, we have recognized that for specifying Signal Generator rules for evaluating the health state of an IR sensor, the values of the neighboring IR sensors and their history have to be taken into account. Furthermore, all combinations of values (of neighboring sensors and history values) are possible to occur in true situations as well as in case of faulty sensors. Consequently, the rules can only reflect probabilities of combinations of values.

For specifying the Signal Generator rules, we have examine potential combinations of sensor values with their according history values and rated each situation by a probability score. The resulting health signal is specified as a score value from 0 to 8 with 0..2 defining *healthy state*, 3..5 defining *suspicious state* and 6..8 defining the *dangerous state*. Any combination of values is scored accordingly:

- if a value combination is in all probability a failure-free situation, the health signal value is decremented by 3 (of course not falling below the value 0). An example for such a situation as (nearly) equal values of neighboring side sensor.

- if a value combination is quite probable to be failure-free, the health signal value is decremented by 1.

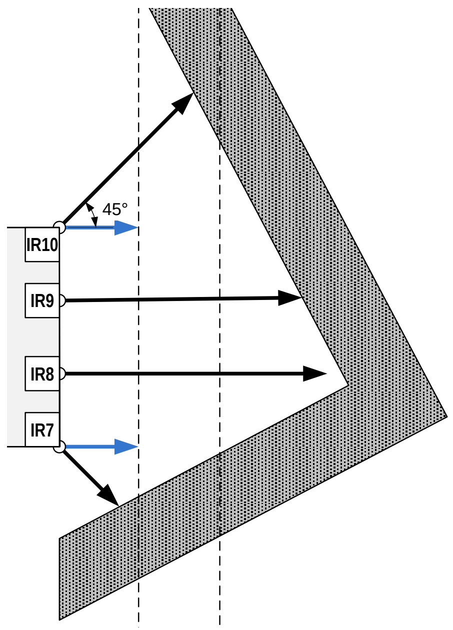


Figure 11.8: Situation for possible combinations of neighboring IR sensor values.

- if a value combination is probable to occur in a case of a present failure and is expected to occur rarely in a failure-free situation, the health signal value is incremented by 1
- if a value combination is very probable to occur in case of a failure, the health signal value is incremented by 3. An example for such a situation has already been described above, if one IR sensor value remain 0 for more than one execution period while the values of the neighboring IR sensors differ.

With these simple rules, we could score any potential position of the BeBot in relation to wall distances based on its probability of occurrence. According to the present value combinations collected by the IR Sensor Monitor, the health signal will evolve. In presence of a failure, the health signal value quickly (in two consecutive execution periods) yields to a *danger signal*. In contrary, value combinations that probably occur in case of a failure even if there is no failure injected will raise a *warning signal*. But as these situations are expected not to persist in further execution periods, the health signal value will be decremented as soon as a value combination occurs that reflects a failure-free situation.

We will not encounter in detail the specified rules that have been implemented as it was not the scope of this thesis to design a reliable IR Sensor Monitor and to prove its quality. Nevertheless, we have evaluated the specified rules in experiments reflecting different situations and thereby refined the rules accordingly. Especially, we have evaluated the specified Signal Generator rules for the IR Sensor Monitor in terms of false alarms. The IR Sensor Monitor did not deliver any *danger signal* if no failure was injected. *Warning signals* have beed output even in absence of failures as, of course, numerous situations exit that produce equal sensor values if a failure is present. However, these *warning signals* have been weakened and eliminated in subsequent execution periods and recovered to a *healthy signal*. Based on these experiments, we have obtained a rather reliable IR Sensor Monitor that can be efficiently applied for the purpose of our evaluation.

11.2.7 Classification Mappings

The BeBot application mainly operates on the system calls related to the BeBot control. However, the execution of these system calls does not only effect the health parameters of the IR Sensor Monitor. According to the classification method (see Chapter 8.7.2) of the OS Health Monitor, the effects of the system calls on the health parameters of all the provided Component Monitor Modules have to be determined. In this section, we present the effected health parameters by the BeBot control related system calls.

1. **Specific system calls under QEMU**

   ORCOS offers specific system calls that encapsulate the communication functionality of an additionally integrated communication module (see Sec.11.1.2 for ACM) which is required - if ORCOS is executed under QEMU - for communication with external systems.

	Scheduler Monitor			
System Call	cOk	cError	Upload	Download
sendto_QEMU	x		x	
recvfrom_QEMU	x	x		x

Table 11.1: Mapping of System Calls to OS Health Parameters: Specific system calls under QEMU

2. **System calls for the BeBot control**

   For the BeBot applications, system calls have been specified to control the BeBot's actions and request the BeBot's sensor data. The BeBot's actions are initiated by the application's decision process mainly governed by the configurations of the values delivered by the IR sensors. At the same time, a system call that initiates any movement of the BeBot provokes changes in the values of the IR sensors.

   The BeBot control related system calls are sent over QEMU through the ACM to the VE. Consequently, each BeBot control related system call uses the communication module and, thereby, effects the according health parameters of the Communication Monitor.

   To provide an adequately structured overview of the correlation between the BeBot control related system call and the effected health parameters, we have categorized the great number of offered system calls according to their analogy in terms of their behavior and their effects on health parameters. In the following, we provide listings of the system calls that effect the same health parameter set.

   (a) **No health parameters:**
   The following system calls do not effect any health parameter by their execution.

   - `getMedian`

   (b) **Communication Monitor:** `Upload`

The following system calls send command messages to the BeBot, but do not expect any response. Therefore, these system calls change only the `Upload` parameter of the Communication Monitor:

- `LEDtoRed`
- `LEDtoGreen`
- `LEDtospecificColor`
- `setMotorSpeed`
- `stopMotor`
- `searchWallState`
- `rotateLeftState`
- `driveStraightWallState`
- `rotateRightState`
- `leavingWallState`
- `connectionclosed`
- `updateState`

(c) **Communication Monitor: `Upload`, `Download`**

The following system calls send command messages to the BeBot to ask for data delivery from the BeBot. Therefore, `Upload` and `Download` parameters of the Communication Monitor are effected by these system calls:

- `readIRSensorValue`
- `getBebotPos`
- `getTargetPos`
- `readIRSensorValue0`
- `readIRSensorValue1`
- `readIRSensorValue2`
- `readIRSensorValue3`
- `readIRSensorValue4`
- `readIRSensorValue5`
- `readIRSensorValue6`
- `readIRSensorValue7`
- `readIRSensorValue8`
- `readIRSensorValue9`
- `readIRSensorValue10`
- `readIRSensorValue11`
- `getCurrentState`
- `getReached`
- `sendUniversalmsg`
- `autodrive`
- `randomautodrive`

From this subset, those system calls that read out IR sensor data also effect the according health parameters of the IR Sensor Monitor. These are:

- `readIRSensorValue` which requests the sensor data of all IR sensors,

- `readIRSensorValue`x requesting the distance value of IR sensor x,
- and `autodrive` and `randomautodrive` that internally request IR sensor data.

(d) **Communication Monitor: `Download`; Memory Manager Monitor: `Alloc Counter`, `Alloc Size`, `Free Counter`, `Free Size`, `Used Memory`**
The following system calls effect the download parameters of the Communication Monitor. Furthermore, memory space is allocated as well as freed again. Therefore, Communication Monitor and Memory Manager Monitor parameters are effected by:

- `bonjour`

Based on the classification mappings, the Signal Generator is able to determine the effected Signal Generator rules have to be processed for classification of the BeBot' application behavior (according to the method described in Chapter 8.7.2) in order to assign the respective health signals.

11.3 Interaction and Control

By using a VE, we have a safe environment by definition. To enable the evaluation of the effectiveness of the Online Anomaly Detection, e.g. in terms of detecting faulty hardware, a full control of both, the VE and the VP, is required. This is realized by integrating interactive control over the VE and the VP at runtime.

First, in order to offer a dynamic environment, the Virtual Evaluation Environment can change the actual maze as well as the type of the maze itself on the fly (by a provided GUI) by a randomized generation and a switching between different algorithms for setting up the maze. It is possible to generate simple maze structures, where all walls are connected, but also structures with one or more loops, so that the pathfinding algorithm of the BeBot must be smarter. Furthermore, we are able to change the size of the maze, the start point of the BeBot and the destination position inside the maze.

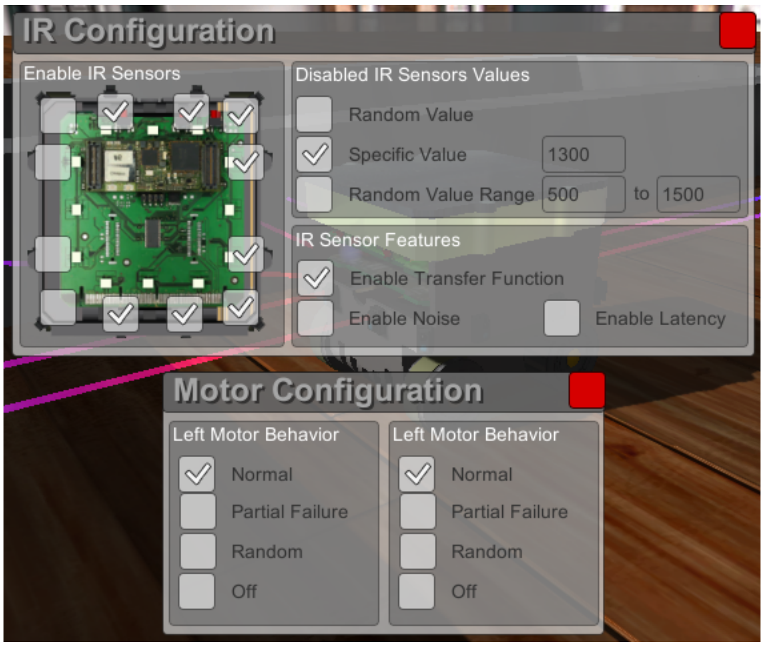
Secondly, as one of the important means, we integrated the possibility to interactively control the data received from the virtual IR sensors and thereby allow failure injection based on the failure model implemented. Single sensors can be completely disabled by the user that can decide which value the defective sensor is sending back to the operating system. It can be a defined value, e.g. 0 or any other static value, random noise and a random value R between two boundaries $[a, \ldots, b]$. Including this method, we are able to selectively inject failures and evaluate the reaction of the Online Anomaly Detection's decision making process.

In order to evaluate the BeBot's execution function, we included a method which manipulates one or both gear motors. We can decide whether a motor will not react at all, randomly react correctly on a driving command or just react randomly whatever value is sent.

For this interactive control, a graphical user interface as shown in Figure 11.9 was designed to change all parameters defined above while the system is in execution. The great benefit using the Virtual Evaluation Environment in contrast of using the real BeBot is the possibility of simulating faulty hardware in various ways. Trying to do that by using the real hardware is hard and may also not include all the options offered by the Virtual Evaluation Environment.

11.4 Evaluation Output

As one essential requirement, the evaluation environment has to provide an output of the evaluation that allows to asses the system performance (see **R.6**) with respect to its experiments'



IR Configuration

Enable IR Sensors

Disabled IR Sensors Values
- [ ] Random Value
- [✓] Specific Value 1300
- [ ] Random Value Range 500 to 1500

IR Sensor Features
- [✓] Enable Transfer Function
- [ ] Enable Noise
- [ ] Enable Latency

Motor Configuration

Left Motor Behavior
- [✓] Normal
- [ ] Partial Failure
- [ ] Random
- [ ] Off

Left Motor Behavior
- [✓] Normal
- [ ] Partial Failure
- [ ] Random
- [ ] Off

Figure 11.9: Graphical user interface for changing the parameters of the Virtual Evaluation Environment.

objective.

Ensuring a reproducible evaluation scenario requires the availability of all (static) information about the environment's configuration as well as the configuration of the executing system. The configuration of the evaluation environment (including maze, start position and destination position) can be extracted from the config files of the Virtual Evaluation Environment. The configuration of the executing system composed of ORCOS including the Online Anomaly Detection and the application tasks is stored in the SCL configuration files.

As a basic output, the Virtual Evaluation Environment provides a visualization of the system execution. Thereby, the BeBot's performance can be observed which allows to draw conclusions on the quality of the BeBot's application. Furthermore, the commands sent by the application to drive the BeBot are displayed so that the application's behavior can be traced online. However, instead of evaluating the BeBot application, the scope of the case study is the Online Anomaly Detection so that we are more interested in the output related to that.

Our evaluation environment provides two major instruments for evaluation output related to the Online Anomaly Detection:

1. **Log Files** The Online Anomaly Detection operates inside ORCOS and has no direct connection to the virtual environment. Hence, information that reflects the execution state is encapsulated within ORCOS. For tracing the execution of ORCOS or its components, logs are output on the execution console. Additionally, the logs of the Online Anomaly Detection are written into log files to enable an analysis of an executed evaluation scenario afterwards. From these log files, the complete execution can be reconstructed and all information required can be extracted:

   - the arriving system calls and their location in the Suffix Tree (with all additional information such as occurrence counter etc.)

- the health signals obtained by the OS Health Monitor, separated for each Component Monitor Module
- the resulting classification marker values for Suffix Tree nodes after proceeding the classification process

The Online Anomaly Detection is working fully decoupled from information of the VE. ORCOS and, in particular, the Online Anomaly Detection have no knowledge about changes happening in the environment or the BeBot. This includes also that the Online Anomaly Detection is not aware of whether a failure in an IR sensor has been induced. However, for analyzing the evaluation scenario, it is important to combine behavior of the Online Anomaly Detection with the states of the VPs in the Virtual Evaluation Environment. Assuming that at a particular point of time the classification outcome changes (e.g. into a suspicious or dangerous state), it is important to know whether this change was caused by a change in one of the objects of the Virtual Evaluation Environment or arose because of an internal system instability. On the other hand, knowing that a failure was induced in a particular evaluation scenario, it is important to examine whether this failure was detected by the IR Sensor Monitor, how long it took to detect this failure (in number of periods) and whether this failure will effect the classification and, hence, the resulting health signal. To overcome this problem, we have implemented a further interface between the VE and ORCOS through which the VE can send a message in case of a user-initiated change. The information in this message is not incorporated to the Online Anomaly Detection at all. It only initiates a notification of e.g. a failure injection into the log file for supplementary analyzing purposes that allow to draw conclusions of the performance of the Online Anomaly Detection in context of the state of its application environment.

The logfiles enable the complete execution to be reconstructed and are the main instrument for assessing the performance of the Online Anomaly Detection with respect to the objective of the evaluation scenario.

2. **Debugging**

   A further aspect of the evaluation is concerned with the question of whether specific states of a implementation (programming code) can be reached during the execution. This could be conditions that are expected to be very uncommon or which are specifically intended. We call them states of interest. Identifying whether such a state is reached on a system that is executing is not easy. However, by executing the system within the Virtual Evaluation Environment with the (application controlling) software executing in the QEMU emulator, it becomes possible to debug the executing code. We exploit the debugging facility to set breakpoints in the programming code (of the Online Anomaly Detection) at particular interesting parts in terms of the evaluation objective, e.g. a produced *danger signal*. When reaching a breakpoint, the execution of ORCOS is interrupted which leads to the fact that the BeBot is brought to a stop as it does not receive any commands during the break. This allows to directly identify - during the execution - whether the particular states of interest were reached. Because of the potentials it offers, the debugging facility is considered as a second major means to produce evaluation output for analyzing the effectiveness of our approach.

The visualization of the system execution provided by the Virtual Evaluation Environment, the debugging facility of the operating system and the log files generated by the Online

Anomaly Detection provide suitable instrument to capture the system performance and to obtain comprehensive evaluation outputs.

11.5 Evaluation Scenarios

For defining evaluation scenarios, we refer to the recommendations of Roske et al. (see Chapter 10.1) to ensure a separation of concerns in terms of evaluating the life cycle phases composed of the perception function, the decision function, and the execution function in an isolated manner (as formulated in requirement **R.4**). For each of the life cycle phases, we clarify the responsibilities of that phase and boundaries to the other phases. Then, we discuss the evaluation objectives of the phase from which we derive evaluation scenarios.

11.5.1 Perception Function

The perception function of the Online Anomaly Detection is composed of system call monitoring and maintaining health signals. This data deals as input for the classification method that represents the decision function. A reliable perception function is required as a prerequisite to evaluate the decision function. Therefore, we have to ensure the correct functionality of the data input sources, namely the System Call Monitor and the OS Health Monitor.

1. **System Call Monitor**

   The system call extraction is directly implemented in the System Call Monitor so that any system call that is executed by an application is recorded with the associated data (system call ID, task ID, time stamp, etc.). This ensures a completeness of the data collection by concept. The correctness of the data extraction has been proven by the implementation that was evaluated in [92]. Based on this, the perception function related to the system call monitoring does not require further evaluation.

2. **OS Health Monitor**

   The OS Health Monitor delivers health signals obtained by processing the Signal Generator rules. Hence, the quality of the health signals relies, on the one hand, on the quality of the rules, and on the other hand, on the data that is processed.

   The data that the Signal Generator rules operate on are the health parameters obtained by the Component Monitor Modules. Health parameters associated with the kernel components replicate the values of the (internal) state parameters of the addressed component or are at least obtained by calculations or accumulations of the available (internal) state parameter values. This data is produced by software, so that the completeness and correctness can be proven by validating the programming code (conducted in [69]).

   The situation is different for the IR Sensor Monitor as this one operates on data delivered by hardware devices. Here, the reliability has to be examined in order to guarantee correctness of the perception function. However, in our evaluation environment the hardware devices are realized in form of virtual prototypes. These, in turn, are implemented as software components that do not suffer from altering effects or defects (if not intentionally induced) and thereby ensure the correct functioning of the perception function.

Furthermore, the implementation of the IR sensor VP offers several alternative models for distance value accuracy as well as for failure injection. These models can be selected in accordance to the requirements of the decision function, which allows a full control over this part of the perception function.

The quality of Signal Generator rules that corporately compute a health signal is strongly depending on the expert knowledge of the system designer. The correct working of the Signal Generator that was implemented for the general kernel modules was shown by performing mutation-based evaluations in [69]. The evaluation of implementation of the Signal Generator defined for the IR Sensor Monitor is described in the evaluation scenario **Scenario 1** (see Section 12.1). These experiments have been performed in a failure-free environment. Although designing a OS Health Monitor component is not the scope of this thesis, it had to be evaluated as its results build up the basis for the evaluation of the performance of the Online Anomaly Detection with failures induced. Without evidence that the IR Sensor Monitor works correctly in a failure-free environment, we cannot analyze and assess results of advanced experiments.

11.5.2 Decision Function

In our Online Anomaly Detection, the decision function, as being the part of the system responsible for analysis, is represented by the classification mechanism in the Anomaly Detection Module and forms the core entity of this approach. For evaluating this, different variations in system behavior have to be initiated.

The Signal Generator was evaluated within the perception function in a failure-free environment. However, resulting *healthy/green* health signals can only lead to *green* classification outcomes that are written into classification markers of the Suffix Tree nodes. The classification method can only deliver classification outputs different from *green* if the OS Health Monitor components return health signals different from *green*. Therefore, the evaluation of the decision function can only be conducted if *warning/yellow* or *danger/red* health signals are raised. This, however, can be generated in a controlled manner if some threats or failures are induced into the system. The easiest way to realize this is to exploit the failure injection provided by the Virtual Evaluation Environment for the IR sensors.

For the purpose of evaluating the qualities of the classification method, we have examined

- **Scenario 2**: how long does it take until the induced failure is detected by the IR Sensor Monitor if it is detected at all (see Section 12.2).

- **Scenario 3**: whether and to what extend the failure effects the classification outcomes of the currently executing application (see Section 12.3).

- **Scenario 4**: the reaction induced by the Online Anomaly Detection based on the outcome of the classification method (see Section 12.2).

The specified scenarios enable to demonstrate all interesting aspects of the classification method. The quality of the classification method is the core issue in terms of the performance of the Online Anomaly Detection so that the evaluation of the decision function is the most significant one.

11.5.3 Execution Function

The evaluation of the execution function is on the one hand concerned with performance parameters, such as runtime or memory consumption. These have been evaluated in the quantitative evaluations that are the topic of Chapter 9.

Another aspect of the evaluation of the execution function is concerned with testing whether the decisions made by the decision function are correctly executed and satisfy the system's objective. In the context of our approach, the execution function concentrates on handling the classification outcomes that, in fact, are passed to the controller (in our application scenario it is the *Controller Task*) that is responsible to decide whether to initiate a system reconfiguration. The passing over of classification outcomes can be verified by programming code validation. The implementation of the controller is not the scope of this thesis and, hence, the reaction on a detected unstable or dangerous state as well. Nevertheless, we have evaluated the initiation of a reconfiguration as one aspect of **Scenario 4**. With this, we could evaluate the offered options for dealing with the Behavior Knowledge Base after a reconfiguration.

11.6 Summary

In this chapter, the evaluation environment for the case study of the Online Anomaly Detection is introduced. The BeBot with autonomous driving applications was chosen to be applied for the case study and transferred into a virtual environment because of the potentials of virtual environments in terms of satisfying the requirements set for the evaluation. The designed Virtual Evaluation Environment is composed of the virtual prototype of the BeBot, the maze as its application environment, the BeBot controlling application and ORCOS on which the BeBot application is executed. To integrate all these system parts into one evaluation environment, some auxiliary techniques have been applied such as QEMU for emulating the ORCOS execution on the BeBot and the ACM for implementing the communication between the virtual environment and ORCOS. The architecture with all applied techniques is introduced in detail.

For the purpose of this case study, the Online Anomaly Detection was extended by a BeBot-related module, the IR Sensor Monitor. It is responsible for monitoring the health states of the BeBot's IR sensors and implements Signal Generator rules for their computation.

The Virtual Evaluation Environment provides means to interactively manipulate the properties of the evaluation entities and, thereby, allows controlled failure injection. This is the most powerful feature for the purpose of our evaluation as it allows to induce intended changes and to evaluate whether and to what extent they will impact the performance of the Online Anomaly Detection, especially, in terms of the outcome of the classification method. The different aspects of the evaluation are formulated in evaluation scenarios that according to the requirements are specified with respect to the separation of concerns.

The application of a virtual environment provides a visualization of the system execution that allows to observe the system performance. The execution states of the Online Anomaly Detection can be traced either at runtime by using the debugging facilities, or after the execution by means of the log files generated during the execution that allow a detailed reconstruction of the execution scenario. These instruments offer a comprehensive evaluation output to allow an analysis of the defined evaluation scenarios.

We define four evaluation scenarios with different objectives that build the basis for the discussion of the results in the next chapter.

CHAPTER 12

Evaluation Results and Discussion

In the former chapter, the entire evaluation environment was described in detail with a short introduction of the evaluation scenarios resulting from the evaluation objectives. This chapter specifies the evaluation scenarios with its basic conditions and parameters and outlines the obtained evaluation results. This chapter is finalized with a discussion of the main results provided by the case study.

Evaluation Configuration

For the evaluation discussed here, the BeBot was executed in the Virtual Evaluation Environment initially controlled by its simple application *left-side wall follower* (see Section 11.2.3). The BeBot applications mainly operate on the IR sensor data values requested by the associated system calls. For the IR sensor behavior, the **LinearModel** (see Section 11.2.2) was applied without any noise or latency. This model was chosen because it enables an easier and more reliable comparison of values delivered by neighboring sensors (as required by the Signal Generator rules for the purpose of health state examination). Even if the *left-side wall follower* application operates only on data from the front and left-side sensors (encompassing $IR0$, $IR1$, $IR2$, $IR3$, $IR4$, $IR10$, and $IR11$), for the evaluation purposes, the application requests all IR sensor data once in a period. This is done in order to supply the IR Sensor Monitor with actual sensor data and to make the values delivered by the IR sensors traceable for the postponed analysis performed here.

As a means to introduce system instabilities in a controlled manner, in this case study, we refer to injection failures performed by the operator/user. The failure injection is achieved on the basis of manipulating the behavior of the IR sensors. The different failure models for the IR sensors (see Section 11.2.2) implement different subsets of Signal Generator rules. On a real system, it cannot be predicted which kind of failure occurs - if one occurs - so that the entire set of Signal Generator rules would have to be processed. However, the evaluation within a safe environment, such as the one applied here, allows us to select a failure model. This, consequently, allows to reduce the set of processed Signal Generator rules.

From the viewpoint of the Signal Generator rules, the failure models **FailureModel1** (IR sensor delivers value 0), **FailureModel2** (IR sensor delivers value max_value) and **FailureModel3** (IR sensor delivers a fixed user-set value) are assigned the same subset of Signal Generator rules. Therefore, **FailureModel1** can be treated as a representative of these three failure models. **FailureModel4** (IR sensor delivers a random value) is assigned a different subset of Signal Generator rules. These rules are more complex as it is not that straightforward to detect a failure if the values delivered by an IR sensor differ in each monitoring period. Because of the different subsets of Signal Generator rules, the experiments in the evaluation scenarios have been conducted by using **FailureModel1** and **FailureModel4** separately in order to allow a comparison of their performance results.

The core entities in this evaluation are the Signal Generator rules of the IR Sensor Monitor. According to Chapter 11.2.6, they mainly operate based on probabilities of combinations of IR Sensor values that score the health signal value. The position of the BeBot towards the present obstacles governs the in IR sensor values. Hence, positions leading to value combinations that are assessed as uncommon assuming the IR sensors are working correctly get a higher probability to declare a suspicion. However, as they may occur even without the presence of IR Sensor failures, they will lead to an increase in the score of the health signal value. In contrary, a failed sensor may be assessed as working correctly because its delivered value may accidentally lay in a range that was assessed as plausible when setting it into context of its neighboring IR sensor values. Because of relying on probabilities, the Signal Generator incorporates a certain degree of inaccurateness and uncertainty. Considering the two different classes of failure models, the Signal Generator rules for **FailureModel1** are more straightforward as a failure in an IR sensor can be rather quickly detected based on examination of the IR sensor's history data (the failed IR Sensor delivers a fixes value either 0, *max_value* or a user-defined fixed value). In contrast to that, in **FailureModel4** a failure can only be detected in the context of the neighboring IR Sensor values as a failed IR Sensor in this mode will deliver values that are different in each period. Because here, the health signal examination is strongly related to the different IR sensor value combinations of neighboring IR sensors, the Signal Generator rules in this failure model are more blurred than those for **FailureModel1** and are therefore assumed to be more inaccurate. Therefore, one objective of this evaluation is to verify the reliability of the IR Sensor Monitor as a prerequisite to evaluate the powerfulness of the Online Anomaly Detection.

It is important to emphasize here, that in the evaluation we can only obtain the operating system internal data and analyze this, meaning the IR sensor values, without any knowledge of the real situation and the position of the BeBot in relation to the maze walls. Therefore, even if getting a health signal score that signals a *warning* or even a *danger*, we are not able to argue based on the environment's current situation. Hence, the reason for particular signals occurring during the experiments cannot be given here. However, we expect false signals that occur because of probably uncommon IR sensor value combinations to be reverted in the following periods as situations causing such signals are expected to be transient.

12.1 Evaluation Scenario 1

As a prerequisite to evaluate the performance of Online Anomaly Detection in case of component failures, it is essential to ensure a certain degree of reliability in case of absence of failures. In this evaluation scenario, we address the reliability of the IR Sensor Monitor in a failure-free environment (as the reliability of all the other modules have been evaluated in [69]). The objective of this evaluation is to verify the correctness of the health signals delivered by the IR Sensor Monitor and their false rate.

We have evaluated the IR Sensor Monitor executing Signal Generator rules for **FailureModel1** and **FailureModel4** within separate experiments with no failure injection performed during system execution. Of course, for each evaluation we have executed multiple experiments. From these, the ones with the most significant results have been selected as representatives for this evaluation scenario, and are presented and discussed here.

1. The IR Sensor Monitor executes the Signal Generator rules assigned to **FailureModel1**. Failures in IR sensors have not been injected.

   A typical result of health signals delivered by the IR Sensor Monitor is illustrated in Fig. 12.1. The IR Sensor Monitor was executed over 300 periods. By evaluating the health state of 12 IR sensors, the IR Sensor Monitor has output $300 \cdot 12 = 3600$ health signals. Almost all health signals are of value 0 which is associated with a *healthy/green signal* and no suspiciones at all. (The health signal of the IR Sensor Monitor is assigned a score value in the range of 0..8, see Section 11.2.6.) In period 85, for one IR sensor ($IR5$), a *warning signal* was raised as the returning health signal value was 3. The value 3 represents a situation that is rather probable to be a suspicion. However, according to our expectations in a failure-free scenario, in the following periods 86 and 87, the score value of the health signal was step-wise diminished. After period 86, the health state immediately recovered to *healthy/green signal* with value 2, continued to diminish with the value 1 in period 87, and, finally, to value 0 afterwards.

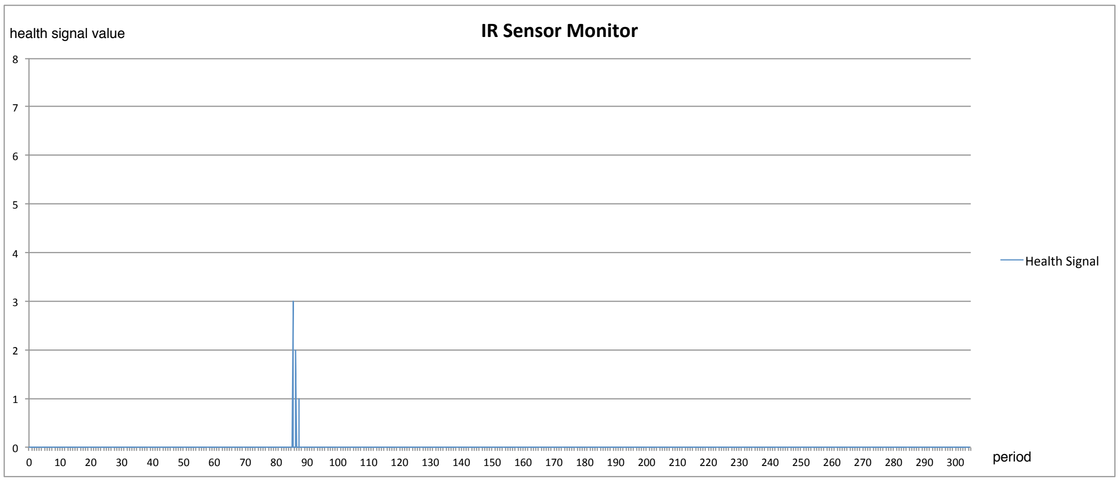


Figure 12.1: Health signals obtained by the IR Sensor Monitor executing Signal Generator rules for FailureModel1 in absence of failures.

   This example illustrates that value combinations of IR sensors exist that represent both, healthy states as well as suspicious situations, and that for the IR Sensor Monitor it is possible to raise a *warning signal* even if no failures were injected as caused by the fact that the Signal Generator rules are defined based on probabilities. Referring to the presented experiment, the false rate is 0.027% with a false alarm case of 1 of 3600 (similar false rates have been obtained in other executions).

   With a false rate of 0.027% and the capability to recover from false alarms, we can state that the IR Sensor Monitor provides an acceptable reliability.

2. Detecting a failure in an IR sensor if the sensor delivers different values in each monitoring period is very complicated. An uncertainty remains whether a particular situation

specified in a Signal Generator rule is a failure or not. Hence, the Signal Generator rules for detecting **FailureModel4** are more unsafe than those for **FailureModel1** as they operate on probability assumptions. This leads to different results for health signals delivered by the IR Sensor Monitor executing the Signal Generator rules assigned to **FailureModel4** (with no failures in IR sensors injected).

The health signals obtained in our experiments are illustrated in Fig. 12.2. In Fig. 12.2, the boundaries between the types of health signals are marked. Furthermore, specific regions or points are marked that will be discussed in more detail later on.

The IR Sensor Monitor was executed over 755 periods evaluating the health state of 12 IR sensors in each period. In this experiment, the IR Sensor Monitor has output $12 \cdot 755 = 9060$ health signals.

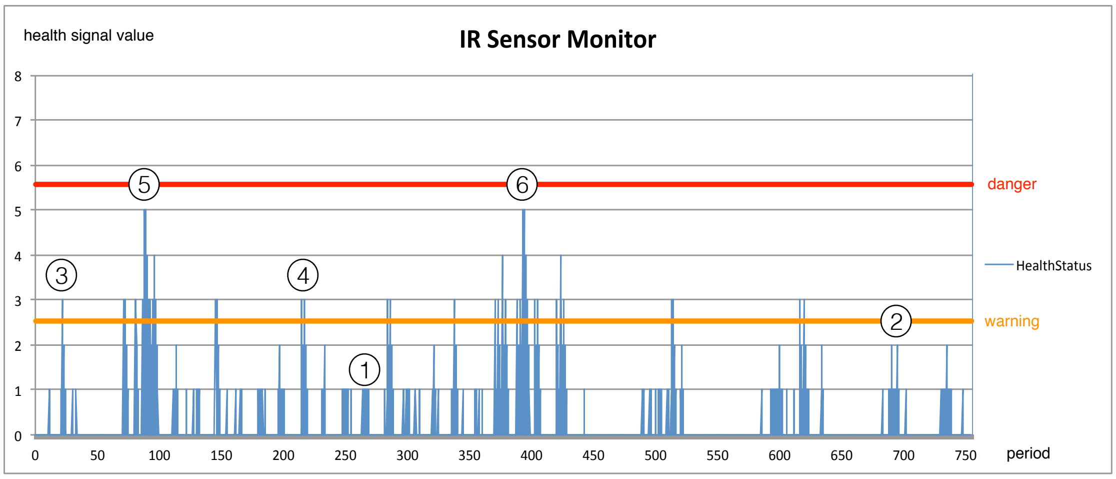


Figure 12.2: Health signals obtained by the IR Sensor Monitor executing Signal Generator rules for FailureModel4 in absence of failures.

From these 9060 health signals, 8745 stay in value 0, while 217 get value 1 and 51 value 2 which both represent still *healthy/green signal*. 47 signals are *warning/yellow signals* with 37 of value 3, 6 of value 4 and 4 of value 5. This leads to a *warning/yellow signal* rate of 0.52%. No *danger/red signal* has been output. Generally, we can state, that all *warning signals* are transient as on the one hand, they do not yield to a *danger signal*. On the other hand, they are able to recover as, after their occurrence, they do not remain in the system, but are eliminated and return back to *healthy/green signal*.

Let us call all health signals whose value differ from 0 *deviators*, even if in case of 1 and 2 they result in the same *healthy/green signal*. (*Deviators* are not considered as anomalies or false signals but only as health signals unequal to 0.) One striking characteristic that can be seen in Fig. 12.2 is that deviators occur in a number of successive periods building up clusters.

Accordingly, the *warning signals* occur in multiple successive periods of a cluster. Generally, this is caused by the fact that a *warning signal* detected in one monitoring period needs some further periods to recover. Hence, instead of considering each of the 47 *warning signals* separately, we are considering the occurrence of *warning signals* within their clusters. In this experiment, we have found 15 clusters with health signals reaching the *warning signal* range.

Based on the log files delivered by the experiments and the results illustrated in Fig. 12.2, we can discuss some detected phenomena and their causes.

1. As already indicated, health signal deviators seem to appear in clusters. Multiple of them can be seen in Fig. 12.2. Marker ① points out one typical situation for such a cluster generation documented in the log files (in this table $IR0$ is located next to $IR11$ to emphasize that these IR sensors are neighbors):

	health signal value							
period	...	$IR4$	$IR5$	$IR6$	$IR7$	..	$IR11$	$IR0$
..	..	..	..	..	..	..	..	..
262	0	0	0	0	0	0	0	0
263	0	0	0	0	0	0	1	0
264	0	0	0	0	1	0	0	0
265	0	0	0	1	0	0	0	0
266	0	0	0	1	0	0	1	0
267	0	0	1	0	0	0	1	1
268	0	1	0	0	0	0	0	0
269	0	0	0	0	0	0	0	0
..	..	..	..	..	..	..	..	..

Because of the interdependencies between neighboring IR sensor values as well as the impact of the previous classification marker value in the evaluation of the health signals (as specified in the Signal Generator rules in Section 11.2.6), such propagations emerge that lead to the already named clusters of deviators. The most interesting highlight in this excerpt is provided by $IR4$ to $IR7$ (and by $IR11$ and $IR0$ on the side). The remaining IR sensors are consolidated by "..." as they all deliver health signal values of 0 that are not of any significance here. In period 264, in the value of $IR7$ a suspicion is detected that increases the health signal value to 1. Even though, the health signal remains in *healthy/green* state it is worth noting this increase. In the next period, for obtaining the health signal value of $IR6$, the health signal values of its neighbors - including $IR7$ which health signal value was increased in the previous period - imply the result of the health signal value of $IR6$ and yields to a propagation of the health signal value: In period 265, the health signal value of $IR6$ becomes 1 (the same happens in period 266). In period 267 the health signal propagates to $IR5$, and finally to $IR4$ in period 268. ($IR11$ and $IR0$ also exhibit such a propagating effect in periods 266 and 267).

This propagating phenomenon can be observed for multiple times in the provided example. The health signal propagation can be a further amplified, and lead to an additional increase of the health signal value. Such a situation is illustrated in the following excerpt of log files reflecting the region in Fig. 12.2 marked by ②.

	health signal value							
period	...	$IR5$	$IR6$	$IR7$	..	$IR10$	$IR11$	$IR0$
..	..	..	..	..	..	..	..	..
687	0	0	0	0	0	0	0	0
688	0	1	0	0	0	1	0	0
689	0	1	0	0	0	1	0	0
690	0	2	1	0	0	0	0	0
691	0	1	0	0	0	0	1	0
692	0	0	0	0	0	0	1	1
693	0	0	0	1	0	0	1	1
694	0	0	0	2	0	0	0	0
695	0	0	0	1	0	0	0	0
696	0	0	0	0	0	0	0	0
..	..	..	..	..	..	..	..	..

In the periods 688 to 691, the health signal value of $IR5$, on the one hand, propagates to $IR6$ (in period 690). On the other hand, the obtained health signal is enhanced by the previous health signal values of $IR5$ leading to the health signal value increase to 2, that in the following periods is step-wise eliminated. The same phenomenon can be observed for $IR7$ in the periods 693 to 695.

2. The following table shows an extract of the log files of the time interval that is marked by ③ in Fig. 12.2. In this table, $IR8$ and $IR9$ are explicitly listed as they exhibit deviators that are significant here, while the other IR sensors are consolidated because they all deliver 0 health signal value.

	health signal value			
period	...	$IR8$	$IR9$	...
..	..	..	..	..
19	0	0	0	0
20	0	0	1	0
21	0	3	2	0
22	0	2	1	0
23	0	1	1	0
24	0	0	0	0
..	..	..	..	..

On the basis of this extract, two observations can be discussed:

The first is identified by the periods 20 and 21. In period 20, a situation for $IR9$ is detected, that increases the health signal value (to 1). One period later, the health signal value of $IR9$ increases again and results in 2. The increase in the health signal value in one period after the other may be caused by the fact, that the situation of the BeBot that was classified as a suspicion may last longer and still be present in that period. Then, the Signal Generator rules effected here amplify the health signal score. Furthermore, a suspicious combination of IR sensor values may effect the neighboring sensors' health states. This can be observed in period 21: the neighboring IR sensor $IR8$, exhibits a health signal value of 3 in that period leading to a *warning signal*. Hence, the suspicion tends to propagate among the neighboring IR sensors.

The second observation is depicted in the periods 21 to 24. In each period, the health signal values decrement step-wise. This behavior resolves the suspicion and recovers the health state of the IR sensors. This situation proves that the *warning signal* was transient and that the suspicious situation either had not existed or was resolved. This particular example illustrates an ideal form of recovery for *IR*8 as in each following period the health signal value is continuously decremented. (It is not always the case for recovery in other clusters containing *warning signals*.)

3. A mutual enhancement of the health signal values of neighboring IR sensors is depicted by the marker ④ referring to the following health signal values:

	health signal value							
period	...	*IR*4	*IR*5	*IR*6	*IR*7	*IR*8	*IR*9	..
..	..	..	..	..	..	..	..	..
280	0	0	0	0	0	0	0	0
281	0	0	0	0	1	0	0	0
282	0	0	0	0	1	0	0	0
283	0	0	0	1	0	0	3	0
284	0	0	1	0	0	0	2	0
285	0	0	1	0	0	3	1	0
286	0	1	0	0	0	2	0	0
287	0	2	0	0	0	1	0	0
288	0	1	0	0	0	0	0	0
289	0	0	0	0	0	0	0	0
..	..	..	..	..	..	..	..	..

Because *IR*9 was is *warning* state with a health signal value of 3 in period 283, it infects its direct neighbor *IR*8 to also raise a *warning signal* two periods later on. Fortunately, both *warning signals* are transient and are able to recover back to health signal value 0.

4. Another example for such a mutual enhancement is pointed out by ⑤, illustrating the following situation (without illustrating in detail, the situation marked by ⑥ is very similar to the one marked by ⑤):

	health signal value									
period	...	*IR*2	*IR*3	*IR*4	*IR*5	..	*IR*9	*IR*10	*IR*11	*IR*0
..	..	..	..	..	..	..	..	..	..	..
80	0	0	3	0	0	0	0	0	0	0
81	0	3	2	0	0	0	0	0	0	0
82	0	2	1	0	0	0	0	0	0	0
83	0	0	1	0	0	0	0	0	0	0
84	0	0	0	0	0	0	0	0	0	0
85	0	0	0	0	0	0	0	0	0	0
86	0	3	0	0	0	0	0	0	0	0
87	0	2	3	0	0	0	0	0	0	0
88	0	**5**	2	0	0	0	0	0	0	0
89	0	4	**5**	0	0	0	0	0	0	0
90	0	3	4	0	0	0	0	0	1	0
91	0	2	3	0	0	0	0	0	2	0
92	0	1	2	1	0	0	0	0	3	0
93	0	0	1	0	0	0	0	1	2	0
94	0	0	0	0	0	0	3	0	1	0
95	0	0	0	1	0	0	4	0	1	0
96	0	0	0	0	0	0	3	0	1	1
97	0	2	0	0	0	0	2	0	0	0
98	0	0	0	0	1	0	1	0	0	0
99	0	0	0	0	0	0	0	0	0	0
..	..	..	..	..	..	..	..	..	..	..

For *IR*2 a health signal value of 3 is output in period 86 which is a *warning signal*. The state of this IR sensor infects the health signal value of *IR*3 which then also delivers a 3 in the following period. In the following periods (period 88 to 91), the health states of these two IR sensors mutually amplify their health signal values in such a manner that they both reach the value 5 (*IR*2 in period 88 and *IR*3 in period 89). Of course, this situation could have been caused by a BeBot position in relation to a wall, that was estimated to seldomly occur and rather specifies a failure situation. Furthermore, depending on the movement of the BeBot and the resulting new position, the assumption of the IR Sensor Monitor (that the according sensors are in failure mode) could have been reinforced in the following monitoring periods.

Obviously, this situation is more critical than those discussed before, as the health signal value almost reaches the *danger signal* range, however without passing over.

Basically, it could be observed that the health signal value 1 occurred in all IR sensors in a rather equally distributed manner, whereas the higher values, especially, those related to *warning signals* were mainly detected in the left-side sensors *IR*2 and *IR*3 and the front-side sensors. This may be caused by the fact that the *left-side wall follower* algorithm predominantly aims to drive the BeBot close to the left-side wall and senses the front side in oder to detect walls. Hence, the distance values of these IR sensors exhibit more variations than all the other IR sensors that most of the time may be in the situation not to sense any obstacle. Therefore, they are more affected in case of uncovering suspicious distance value combinations which leads to the raised *warning* health signals.

The experiments for evaluating the Signal Generator rules assigned for **FailureModel4** have revealed some interesting phenomena. In comparison to the evaluation in context of **FailureModel1**, the IR Sensor Monitor has delivered multiple *warning signals*. But it was also shown that all of them have been resolved in the following periods. In the executed experiments, no *danger signals* have occurred but it cannot be completely excluded that it might happen due to the mutual propagating effects.

Applying different subsets of Signal Generator rules leads to different results even in a failure-free evaluation. Even though a number of *warning signals* was raised, they all have been eliminated and the health signal values could recover to their base value 0. However, it is important to emphasize, that *warning signals* are suspicions but no real threats. Therefore, we cannot really talk about a false rate as no *danger signals* falsely occurred. Consequently, based on these experiments, we can state that the IR Sensor Monitor is appropriately reliable for the purpose of this evaluation.

12.2 Evaluation Scenario 2

After having ensured an acceptable reliability of the IR Sensor Monitor in a failure-free environment, it is essential to examine the performance of the IR Sensor Monitor in case of failures as it is the major input-delivery module for the Online Anomaly Detection in the context of this case study. To evaluate this, we have performed experiments with the same configurations as described in **Scenario 1** but with actively inducing failures in the IR sensors.

A failure to an IR sensor can be injected by the user in a comfortable way by using the GUI provided by the Virtual Evaluation Environment (introduced in Section 11.3). The failed IR sensors keeps returning that value determined by the applied failure model as long as the failure persists. If the user unmarks the set failure in the GUI, the according IR sensor is recovered and returns to function correctly.

The main questions to be answered by these experiments are:

- Will an induced failure be detected by the IR Sensor Monitor?
- How long does it take to detect the induced failure (in number of monitoring periods)?
- Will a failure - if detected - lead to a *danger signal* or will the IR Sensor Monitor only raise *warning signal*?

To be able to answer these questions, the information must be incorporated into the log files that form the basis of the evaluation output. However, ORCOS is responsible to write the log files and is completely decoupled from information or configurations related to the Virtual Evaluation Environment. In order to supply ORCOS with the information of injected failures, a further interface was implemented (as already introduced in section 11.4) that signals the failure injection to ORCOS. When receiving this signal, a marker is written into the log files that determines the point of time of a failure injection (with a small but acceptable latency). By this, it is possible to derive the monitoring period of the IR Sensor Monitor at the point of time the failure injection and enables to determine the detection period of the failure in forms of the *danger signal*. We have extended the provided log files by a further column for inserting the detection period.

At any point of time, there is maximally one IR sensor suffering from an injected failure. The failure is present in the system until it is detected (by means of delivering a *danger signal*). By use of a breakpoint at the code fraction that determines the part when the IR Sensors detects a *danger signal* we could make sure that the user will not add further failures to the system as long as the present one is not detected (formulated in evaluation instructions). When the breakpoint is reached, the system execution is suspended which signals the detected failure to the user. Only then, the user is allowed to eliminate the failure source by unmarking the IR sensor, before inducing the next one.

Again, we split the discussion of the results into experiments that apply Signal Generator rules for **FailureModel1** and for **FailureModel4**.

1. If a failure is injected by applying **FailureModel1**, the distance value delivered by the according IR sensor is set to 0. The most critical case provided by this failure model is that the distance value 0 represents the case of no obstacle is being sensed in the range of the sensor. Hence, if no obstacles are located in the range of that side of the BeBot that suffers from the sensor failure, it is not that straightforward to detect this failure.

   We have performed multiple experiments to evaluate the failure detection performance of the IR Sensor Monitor, but reduce the discussion on the results of the most striking phenomena:

   1. A failure was induced in the IR sensor *IR*3 in period 225. The following excerpt of the log file is related to this failure injection:

	health signal value			detection period
period	...	*IR*3	...	
..	..	..	..	
225	0	0	0	
226	0	3	0	
227	0	6	0	2
228	0	8	0	
229	0	8	0	
...	..	8	..	
233	0	8	0	
234	0	0	0	
...	..	..	..	

      The failure was immediately detected, as in the period 226 right after the failure injection, the IR Sensor Monitor raised a *warning signal* (health signal value 3) for sensor *IR*3. In the next period, this health signal value was enhanced and resulted in a *danger signal* with health signal value 6. In this case it took 2 periods of the IR Sensor Monitor to identify the failure. In the following periods, the health signal value was even reinforced up to value 8 and remained until period 233 - the length of time the failure was present. After deactivating the failure source in the GUI, the health signal value of *IR*3 was recovered immediately in the next monitoring period to the basic value 0. The health signal values of the remaining sensors are not affected here. This example points out the accuracy of the Signal Generator rules for

FailureModel1: no health signal value propagation is emerging in this mode, as the Signal Generator mainly operates on the particular IR sensor's own (history) data.

2. The following log file excerpt shows that a failure is detected quickly, but is recovering step-wise:

	health signal value			detection period
period	...	$IR4$	...	
..	..	..	..	
273	0	0	0	
275	0	3	0	
276	0	6	0	2
277	0	8	0	
...	..	8	..	
280	0	8	0	
281	0	7	0	
282	0	6	0	
283	0	5	0	
284	0	4	0	
285	0	3	0	
286	0	0	0	
...	..	..	..	

In this example, the failure was immediately detected in period 275 (2 periods for detecting and raising the *danger signal*) after its injection in period 273. The failure persisted in the system for several periods until it is eliminated in period 280. However, the recovery from the reached health signal value 8 to 0 happens in a step-wise decrease. This may be caused by the BeBot's position in relation to potential obstacles in the BeBot's range.

3. Another interesting highlight is depicted by the following excerpt:

	health signal value			detection period
period	...	$IR0$	...	
..	..	..	..	
181	0	0	0	
182	0	3	0	
183	0	6	0	62
184	0	8	0	
185	0	7	0	
186	0	0	0	
...	..	..	..	

The front-side sensor $IR0$ is affected by a failure injection. The BeBot drives by following the left-side wall and senses the front-side sensors to prevent to drive against a wall. Moving forward requires a obstacle-free driveway. Hence, at any point of time the BeBot drives forward, the front-side sensors shall deliver distance values equal to 0 which is from the viewpoint of the BeBot the most common situation. This leads to the fact that the failure injected in one of the front-side sensors is not immediately detectable, as all front-side sensors return distance values

0 (when the BeBot is allowed to drive). Only if an obstacle occurs, the distance values of the front-side sensors will vary. Then, a detection of a failure becomes possible as it happened in the example illustrated above: It took 62 periods until the IR Sensor Monitor raised a *warning signal* as maybe there were no obstacles on the driveway of the BeBot before. This signal was immediately transferred in the following period to a *danger signal* and recovered after the failure was eliminated by the user.

Based on the experiments performed, we can state that by applying the Signal Generator rules assigned for **FailureModel1** all injected failures have been detected. The IR Sensor Monitor determined the failed IR sensor correctly. No neighboring IR sensors have been affected or falsely classified. However, in some cases it took a while until the failure could be discovered because some failures were not perceptible immediately due to the BeBot position in relation to the maze objects.

2. The evaluation results obtained in the context of the **FailureModel4** will be discussed based on an example that was executed over 357 periods. In this time, the IR Sensor Monitor delivered $12 \cdot 357 = 4284$ health signals. During this execution, 14 failures were injected in diverse IR sensors: a failure was only injected if the previous one was detected and eliminated. So, at any point of time only one IR sensor failure was present. Basically, we can state that all failures have been detected by the IR Sensor Monitor. In the following, we discuss some striking phenomena:

   1. In some cases, the failure was immediately detected within 2 or 3 monitoring periods. Here are the excerpts of the log files:

	health signal value					detection period
period	...	$IR9$	...	$IR11$	$IR0$	
..	..	..	..	..	..	
18	0	0	0	0	0	
19	0	1	0	1	0	
20	0	4	0	0	1	
21	0	7	0	0	0	2
22	..	8	..	..	..	
...	..	..	..	..	..	

The failure in the IR sensor $IR9$ was detected after 2 periods.

	health signal value					detection period
period	...	$IR2$	...	$IR11$	$IR0$	
..	..	..	..	..	..	
181	0	2	0	0	0	
182	0	0	0	0	3	
183	0	0	0	0	6	2
184	0	0	0	1	5	
185	0	0	0	0	5	
186	0	0	0	0	8	
187	0	0	0	0	8	
...	..	..	..	..	..	

The failure injected into IR sensor $IR0$ was also detected after 2 periods. In the following period, the health signal value was weakened and then again reinforced in the later periods.

2. Some failures have been quickly discovered, however, it took some more periods until the *danger signal* was raised:

	health signal value									detection period
period	...	$IR4$	$IR5$	$IR6$	..	$IR8$	$IR9$	$IR10$	...	
..	..	..	..	..	..	..	..	..	..	
30	0	0	0	0	0	3	1	0	0	
31	0	0	0	0	0	2	0	1	0	
32	0	1	0	0	0	1	0	1	0	
33	0	0	0	0	0	4	0	0	0	
34	0	0	0	1	0	4	1	0	0	
35	0	0	0	0	0	7	0	0	0	5
36	0	0	1	0	0	8	0	0	0	
...	..	..	..	..	..	..	..	..	..	

In the first period after the failure injection, the IR Sensor Monitor raised a *warning signal* for the affected $IR8$. However, it took 5 periods until the health signal evolved into a *danger signal*. The health signal of $IR8$ even slightly recovered before it was reinforced. One reason for this might be, that the distance value returned by $IR8$ in that period accidentally lay in a range of plausible values when set in the context of its neighboring IR Sensor values (as randomly generated). Similar situations with detection periods of 5 or 6 could be observed for several times in the experiments leading to 5 or 6 detection periods.

3. In some cases, the failure detection required more time, such as:

	health signal value											detection period
period	...	$IR2$	$IR3$	$IR4$	$IR5$	$IR6$	$IR7$	$IR8$	$IR9$	$IR10$	...	
..	..	..	..	..	..	..	..	..	..	..	..	
142	0	0	2	0	0	3	0	0	0	0	0	
143	0	0	1	0	0	2	0	0	0	0	0	
144	0	0	0	1	0	1	0	0	0	0	0	
145	0	0	0	0	0	0	0	0	0	0	0	
..	..	..	..	..	..	..	..	..	..	..	..	
154	0	0	0	0	0	0	0	0	0	0	0	
155	0	0	0	0	0	0	0	1	1	0	0	
156	0	0	0	1	0	3	0	0	0	0	0	
157	0	0	0	0	0	4	0	0	0	1	0	
158	0	0	0	0	0	7	1	0	0	1	0	16
...	..	..	..	..	..	..	..	..	..	..	..	

The failure injected into $IR6$ was immediately signalized by a *warning period* in period 142 right at its injection. For the periods 145 to 155 the health signal of the sensor completely recovered even if the failure was still present. Finally, in period 156 the IR Sensor Monitor raised a *warning signal* again that quickly evolved to a *danger signal* in period 158. The detection of the failure by raising a *danger signal* required 16 monitoring periods. One possible explanation for this phenomenon (health signal recovery) is that $IR6$ is located on the back side of the *BeBot*. The back side sensors, however, are not really engaged in the BeBot's sensing and decision making process except for the situation when the BeBot rotates (it checks the back side sensors to prevent hitting a wall).

However, this phenomenon can also happen to other sensors such like $IR3$ located on the left side:

	health signal value							detection period
period	...	$IR3$	...	$IR7$	$IR8$	$IR9$	$IR10$	
..	..	..	..	..	..	..	..	
218	0	3	0	0	1	0	0	
219	0	2	0	0	0	0	0	
220	0	1	0	0	0	0	0	
221	0	0	0	0	0	3	0	
222	0	0	0	0	0	2	0	
223	0	0	0	0	1	1	0	
224	0	0	0	0	0	0	0	
225	0	0	0	1	0	0	1	
226	0	3	0	0	0	0	1	
227	0	6	0	0	0	0	0	9
...	..	..	..	..	..	..	..	

The failure was injected in period 218 in sensor $IR3$. $IR3$ is a left-side sensor and a sensor on which the BeBot's applications relies on. At the time of injection, the IR Sensor Monitor raises a *warning signal* of health signal value 3 which is weakened in the following periods until it completely recovers to the basic value 0 in period 222. Then, in period 226 it is detected again and yields in a *danger signal* with health signal value 6 in period 227. The complete detection phase takes 9 periods. One reason for the interim recovery of the health signal value is that the random distance values generated by **FailureModel4** might accidentally be similar to the distance values of the neighboring sensors and are therefore estimated to be correct by the Signal Generator rules.

4. In another case, it took even a long time of 76 periods for failure detection as the failure was again injected on a back-side sensor. We have reduced the excerpt of the log file to the most significant lines, only considering the effected IR sensors $IR5$:

	health signal value			detection period
period	...	$IR5$	...	
..	..	..	..	
272	0	3	0	
273	0	2	0	
274	0	1	0	
275	0	0	0	
..	..	..	..	
284	0	3	0	
285	0	2	0	
286	0	1	0	
287	0	0	0	
..	..	..	..	
316	0	3	0	
317	0	2	0	
318	0	1	0	
319	0	0	0	
..	..	..	..	
346	0	3	0	
347	0	2	0	
348	0	5	0	
349	0	8	0	76
...	..	..	..	

The failure in $IR5$ was injected in period 272 and was finally detected in period 349. In the meantime, $IR5$ was assigned a health signal of *warning* for multiple times that could recover.

This example shows that failures can lie dormant in the system without being detected by means of raising a *danger signal* for a long period of time. But, nevertheless, there is a point of time when they are definitely detected.

The evaluation of the IR Sensor Monitor with failure injection by applying **FailureModel4** has shown that all injected failures could be detected. Hence, we can assume that the performance of the IR Sensor Monitor is appropriately reliable.

Random distance values delivered by a failed sensor, of course, can lie in a value range that is assessed to be correct in relation to the other distance values. Even if some failures could lie dormant in the system for a while, they finally all have been detected. Basically, failures in IR sensors that the BeBot application mainly operates on, could be detected quickly within 5 or even 2 periods. Failures in those IR sensors that are requested only in special cases (such as back side sensors for turning) were able to persist in the system as long as they were not required for the decision making process of the application. But, when becoming relevant for the application, the failure was detected and a *danger signal* was raised by the IR Sensor Monitor.

We could see by these experiments, that the IR Sensor Monitor did not raise false health signals as no *danger signal* was raised without a failure being injected.

Of course, this is a case study and not an evaluation with statistical significance, but for the purpose of this evaluation it provides adequate results. The evaluations for **FailureModel1** have outlined that a failure can be detected rather quickly whereas for **FailureModel4** the detection quality is more diverse. However, with these examples we could show the reliability of the IR Sensor Monitor that delivers input for the Online Anomaly Detection.

12.3 Evaluation Scenario 3

In the previous evaluation scenarios, we have ensured an appropriate reliability of the IR Sensor Monitor in a failure-free mode as well as in case of a failure. However, the IR Sensor Monitor is only one of the modules that provide input to the classification method of the Online Anomaly Detection. In this evaluation scenario, the main objective is concerned with the effects on the classification outcome obtained based on the health signals delivered by the OS Health Monitor component modules when classifying executed system calls that are represented as nodes in the Suffix Tree.

The verification of the correctness of the classification outcome includes the following questions:

1. Which classification outcome is produced in a failure-free execution? Which classification outcome is delivered in case of a failure in one system entity?
2. How can failures be detected as soon as possible? Can a failure be identified before one of the OS Health Monitor components raises a *danger signal*?
3. Can such *warning signals* delivered by the IR Sensor Monitor in failure-free modes be masked by the Online Anomaly Detection and finally result in a global *healthy/green signal* or do they lead to incorrect global health signals?
4. Can failures that lie dormant in the system be detected before an action is performed that relies on the failed entity?

Answering these questions requires a consideration of the classification outcome that is composed of the sub-signals output by the single Component Monitor Modules. The health states of the single Component Monitor Modules are recorded in the log files whenever the component's health state was evaluated. We assume that all Component Monitor Modules are reliably based on [69] that provides the evaluation of the operating system-related components by means of mutation testing and on the results offered by **Scenario1** and **Scenario 2** of this evaluation that examined the reliability of the IR Sensor Monitor.

Different detection precisions can be implemented in the Online Anomaly Detection that will trigger an alert to the Controller in order to induce a reaction (see Chapter 8.8.5: one *warning signal*, subsequent *warning signals* in one component, *warning signals* in multiple components at the same time, or at latest one *danger signal*). In our case study, we have to deal with an imprecise IR Sensor Monitor. Hence, the detection precision is strongly depending on the combination of health signals delivered by the other Component Monitor Modules. Therefore, we have configured the Anomaly Detection Module to signal an alert if multiple *warning signals* spread out through different Component Monitor Modules are present. This enables the Online Anomaly Detection to raise an alert as early as possible.

For the experiments in a failure-free mode, all components are expected to be in a *healthy/green* state. This assumption could be proven by the executions of the example scenarios, mainly by taking a view on their log files. We use a log file excerpt for illustrating this. Therefore, we use the first example of **Scenario1** provided for the **FailureModel4** evaluation and extend it by the health signals obtained by the Component Monitor Modules of the OS Health Monitor stored in the log files. (In the tables **H** is used for *healthy/green signal*, **W** is represents the *warning/yellow signal* and **D** the *danger/red signal*.)

	health signal value								detection period	IR Sensor Monitor	Communication Monitor	Scheduler Monitor	Processor Utilization Monitor	Memory Monitor
period	...	$IR4$	$IR5$	$IR6$	$IR7$	..	$IR11$	$IR0$	-	-	-	-	-	-
..	..	..	..	..	..	..	..	..		H	H	H	H	H
262	0	0	0	0	0	0	0	0		H	H	H	H	H
263	0	0	0	0	0	0	1	0		H	H	H	H	H
264	0	0	0	0	1	0	0	0		H	H	H	H	H
265	0	0	0	1	0	0	0	0		H	H	H	H	H
266	0	0	0	1	0	0	1	0		H	H	H	H	H
267	0	0	1	0	0	0	1	1		H	H	H	H	H
268	0	1	0	0	0	0	0	0		H	H	H	H	H
269	0	0	0	0	0	0	0	0		H	H	H	H	H
..	..	..	..	..	..	..	..	..		H	H	H	H	H

In this example, all IR Sensor Monitor health signals are *healthy/green* as well as the health signals of the remaining components. This example is representative for all the other parts of the experiments' log files with health signal of *healthy/green* for the IR Sensor Monitor, as all the operating system-related health signals remain *healthy/green* all the time during the execution.

However, we have seen that the IR Sensor Monitor also outputs *warning/yellow signals* (using Signal Generator rules for **FailureModel4**) even if no failure is present. An example is offered by the second log file excerpt of **Scenario1** that we extended here by the health signals recorded in the log files:

	health signal value				detection period	IR Sensor Monitor	Communication Monitor	Scheduler Monitor	Processor Utilization Monitor	Memory Monitor
period	...	*IR8*	*IR9*	...	-	-	-	-	-	-
..	..	..	..	..		H	H	H	H	H
19	0	0	0	0		H	H	H	H	H
20	0	0	1	0		H	H	H	H	H
21	0	3	2	0		W	H	H	H	H
22	0	2	1	0		H	H	H	H	H
23	0	1	1	0		H	H	H	H	H
24	0	0	0	0		H	H	H	H	H
..	..	..	..	..		H	H	H	H	H

Besides the IR Sensor Monitor's health signal that outputs a *warning/yellow signal*, the remaining OS Health Monitor components have returned a *healthy/green signal* all the time. Again, this example is representative for the remaining parts of the experiments: even though the IR Sensor Monitor returns a *warning/yellow signal*, all the other health (sub-)signals are *healthy/green*.

In this case, we know that there is no failure existing in the IR sensor and hence the resulting *warning signal* can be considered as false. Because of knowing that the IR Sensor Monitor is not strictly precise (in terms of *warning signals*), one central question is what is the classification outcome and which value shall be set to the classification marker of the according system call? (The IR Sensor Monitor is more imprecise when applying Signal Generator rules for **FailureModel4**. Therefore, the results discussed here have been obtained in the context of experiments applying Signal Generator rules for **FailureModel4**.) As already shown in **Scenario1**, in the presented example, the health signal recovers quickly and returns to *healthy/green signal* back again. But this can only be verified by awaiting the following periods in which the health signal is either dissolved or it reveals a failure if it switches to *danger signal*.

Therefore, one main challenge is how to distinguish between a valid *warning signal* and a false one, without being forced to await the subsequent executions. The ability to immediately classify a health signal correctly is a means to prevent false classification results. Related to this, the challenge is whether failures can be detected earlier than when raising a *danger signal*. Both challenges have been addressed and realized by our Online Anomaly Detection approach which will be shown in the following.

First, it is important to take into account what happens in case of a failure: Failures are assumed to propagate in a system, meaning that a failure in one system component will have effect on other components that are interdependent. Considering the ORCOS system calls, they often have effect on health parameters of multiple Component Monitor Modules (as it was shown in Chapter 8.7.2). If executing a system call that operates on a failed system entity, the health parameters effected by that system call (belonging to the diverse Component Monitor Modules) will get assigned values reflecting the health state. Then, the failure in the system entity can potentially be implicated in the health parameters of all effected Component Monitor Modules that output an according health signal.

In our case study, the BeBot-related system calls effect health parameters of the IR Sensor Monitor and the Communication Monitor, as all BeBot-related commands are sent over ACM to the BeBot's virtual prototype. The IR sensors of the BeBot are assumed to be controlled by simple microcontrollers that process and return the distance values. In case of a failure, the processing time of the microcontroller is increased, which effects the response time of the according request to the BeBot that in turn increases the response time of the IR sensor-related system calls. In our implementation, we simulate the increase of the microcontroller's processing time by increasing the communication time (over ACM) as introduced in Chapter 11.2.2. This results in an increase of the response time of the request sent to the BeBot and again effects the execution time of the system call and, consequently, of the application itself. Hence, a failure in an IR sensor propagates in such a manner that it implies the health parameters of the IR Sensor Monitor, the Communication Monitor as well as the Scheduler Monitor.

Referring to the log file excerpt above, in which the Component Monitor Modules - apart from the IR Sensor Monitor - have returned a *healthy/green signal*, it points out that no other Component Monitor Modules are in a suspicious health state. This indicates that the *warning signal* of the IR Sensor Monitor might be false. Hence, the *warning signal* delivered by the IR Sensor Monitor is weakened and leads to a *healthy/green* global health signal for that classification entity, illustrated by:

	health signal value				detection period	IR Sensor Monitor	Communication Monitor	Scheduler Monitor	Processor Utilization Monitor	Memory Monitor	Global Health Signal
period	...	*IR8*	*IR9*	...	-	-	-	-	-	-	-
..	..	..	..	..		H	H	H	H	H	H
19	0	0	0	0		H	H	H	H	H	H
20	0	0	1	0		H	H	H	H	H	H
21	0	3	2	0		W	H	H	H	H	H
22	0	2	1	0		H	H	H	H	H	H
23	0	1	1	0		H	H	H	H	H	H
24	0	0	0	0		H	H	H	H	H	H
..	..	..	..	..		H	H	H	H	H	H

The Anomaly Detection Modules examines health signals within their context, taking into account the health signals of the other Component Monitor Modules. By this, the Online Anomaly Detection can more efficiently detect failures, if they are present, at an early point of time.

	health signal value			detection period	IR Sensor Monitor	Communication Monitor	Scheduler Monitor	Global Health Signal
period	...	*IR3*	...	-	-	-	-	-
..	..	..	..		H	H	H	H
127	0	0	0		H	H	H	H
128	0	3	0		W	W	W	W
129	0	2	0		H	W	W	W
130	0	1	0		H	W	W	W
131	0	0	0		H	W	W	W
132	0	3	0		W	W	D	D
133	0	6	0	6	D	W	D	D
134	0	5	0		D	W	D	D
...	..	..	..		-	-	-	-

In this example, the IR sensor $IR3$ raises a *warning signal* in period 128. This signal is enhanced by the fact that the Component Monitor Modules effected by the IR sensor-related system call, the Communication Monitor and the Scheduler Monitor, also show up suspicious health states. By this, the identification of a failure is proven with a high probability so that the *warning signal* of the IR Sensor Monitor is being confirmed and leads to a global *warning signal*. Even though the health signal of $IR3$ is diminished in the following periods, the Communication Monitor and the Scheduler Monitor continue to remain in *warning* health state which indicates that the threat is still present. In period 132, the Scheduler Monitor turns to deliver a *danger signal*. This *danger signal* arises even before the IR Sensor Monitor detects the danger. Hence, the alert is raised at period 128 with a detection period of 1 instead of waiting 6 periods until the failure is definitely identified.

In this stage of the evaluation, we are interested in continuing the execution of the application in order to be able to trace the health signal evolvement. Therefore, we have deactivated the Controller for this evaluation scenario in order to prevent a reaction as this would abort the execution of the application immediately.

For failures that lie dormant, the classification method is even more significant. To illustrate this, we refer to an example from **Scenario2** where it took 16 periods for definitely detecting the failure:

	health signal value									detection period	IR Sensor Monitor	Communication Monitor	Scheduler Monitor	Global Health Signal
period	...	*IR*2	*IR*3	*IR*4	*IR*5	*IR*6	*IR*7	*IR*8	...	-	-	-	-	-
..	..	..	..	..	..	..	..	..	..		H	H	H	H
142	0	0	2	0	0	3	0	0	0		W	W	W	W
143	0	0	1	0	0	2	0	0	0		H	W	W	W
144	0	0	0	1	0	1	0	0	0		H	H	W	W
145	0	0	0	0	0	0	0	0	0		H	W	W	W
..	..	..	..	..	..	..	..	..	..		H	W	W	W
154	0	0	0	0	0	0	0	0	0		H	W	W	W
155	0	0	0	0	0	0	0	1	1		H	D	H	D
156	0	0	0	1	0	3	0	0	0		H	D	W	D
157	0	0	0	0	0	4	0	0	0		W	W	H	W
158	0	0	0	0	0	7	1	0	0	16	D	W	H	D
...	..	..	..	..	..	..	..	..	..	-	-	-	-	-

In period 142, the IR Sensor Monitor delivers a *warning signal* for IR sensor *IR*6. The effected Component Monitor Modules also signal *warning* health states. At this point of time, the Anomaly Detection Module raises a *warning signal* as the global system health state. Sending an alert to the Controller will induce a proper reaction of the detected failure.

In contrary, if only relying on the single health signal of the IR Sensor Monitor, the application would continue executing until the *danger signal* is raised. If failures are lying dormant and remain undetected, there is always a risk that decisions made by the application in an autonomous manner may be false as the application is operating on sensor values that are expected to be reliable although they are false. Here, the Online Anomaly Detection comes into action and prevents incorrect decisions.

By classifying each system call individually which is represented as a node in the Suffix Tree with an own classification marker, the Online Anomaly Detection is able to identify the source of failure. Here, we have to emphasize that a system call related to an IR sensor request also triggers the Communication Monitor to evaluate its health state. This health state is incorporated into the classification marker of the according node. In contrary, the Scheduler Monitor can only be triggered at the point of when the applications completes the execution of its tasks instance. Nevertheless, a *warning signal* in the IR Sensor Monitor in combination with a *warning signal* in the Communication Monitor in one Suffix Tree node is sufficient enough to make assumptions on the source of failure.

With this evaluation scenario, we have demonstrated that the Online Anomaly Detection has the potential of being able to detect failures as early as possible. By considering application behavior in the context of the health signals of the effected Component Monitor Modules, it offers potentials to identify threats even in case of uncertain failure detectors, such as the IR

Sensor Monitor for the **FailureModel4**. By this, it provides potentials to react on failures earlier, maybe even before they are able to propagate throughout the system, as if being undetected they might cause critical states of the entire operating system that could even effect other applications that are executing.

12.4 Evaluation Scenario 4

The Online Anomaly Detection is designed for reconfigurable real-time operating systems. One major ingredient is the individual classification of each occurring system call based on the health parameters by which also unknown behavior patterns are able to be classified. Its detection capability was shown in the previous evaluation scenario. The second ingredient is the Behavior Knowledge Base realized by Suffix Trees that stores all behavior patterns with their associated classifications and is continuously extendable by novel behavior patterns that may be produced because of a system reconfiguration. The objective of this evaluation scenario is to discuss the case of reconfiguration particularly with regard to the Suffix Tree. The Online Anomaly Detection offers different options for the Suffix Tree after the reconfiguration: either to use the same Suffix Tree and extend it by the novel behavior or to build up the Suffix Tree from scratch. We have compared both approaches within this evaluation scenario.

As an example for this, we use the BeBot application of this case study and predominantly refer to the two opposed strategies implemented: *left-side wall follower* and *right-side wall follower*. While the *left-side wall follower* operates on the distance values of the left-side IR sensors (and the front-side sensors to avoid a collision) without taking into account the right-side IR sensors, the *right-side wall follower* uses the right-side IR sensor without any knowledge about the left-side sensors. This means that a failure of an IR sensor on the left-side cannot effect the application performance of *right-side wall follower* (for the *left-side wall follower* vice versa).

In this evaluation scenario, initially the BeBot was controlled by *left-side wall follower*. After a while, a failure in one of the left-side sensors ($IR3$ requested by system call with ID 59) was injected in order to be detected by the Online Anomaly Detection (as presented in **Scenario3**). In that case, the Online Anomaly Detected is responsible to send an alert to the *TaskController* (see Fig. 11.7 in Chapter 11.2.3). The alert is including information on the system call ID that is assumed as the source of failure so that the *TaskController* is able to select another strategy that is not relying on that system call. In case of a failure on the left-side IR sensor, *TaskController* was forced to reconfigure the application to execute *right-side wall follower* as it does not operate on any left-side sensors (no other option was available).

This evaluation scenario was executed with the Behavior Knowledge Base configured to

1. build up a new Suffix Tree from scratch
2. use the same Suffix Tree

after the reconfiguration.

We have made the following observations: Both application implementations exhibit similar behavior as they have a common system call control flow as illustrated by Fig. 12.3 at the beginning of their periods and at the end. Only the middle part of the system call control flow differs. (The highlighted arrows emphasize the most common execution path.) Both application implementations start with executing system call 55 and exhibit equal behavior until system call 59. Then, there is a difference in behavior between both implementations: The control flow of *left-side wall follower* is illustrated in Fig. 12.4 and control flow of *right-side wall follower* is shown in Fig. 12.5. Afterwards, both implementations again execute the same

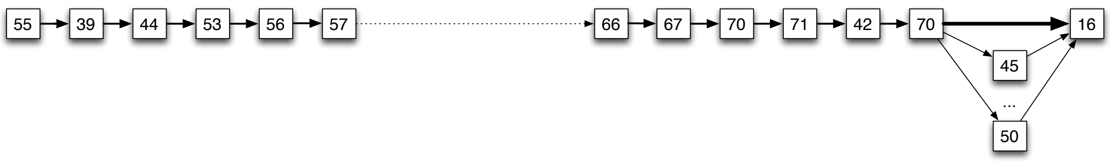


Figure 12.3: Common system call control flow of applications *left-side wall follower* and *right-side wall follower*

system call control flow as illustrated by Fig. 12.3, continuing with system call 66 until the end of the sequence, completing its execution by system call 16 (Sequence Ending Symbol).

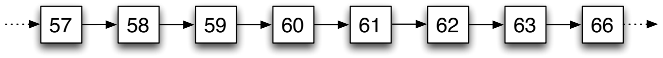


Figure 12.4: Specific system call control flow of *left-side wall follower*

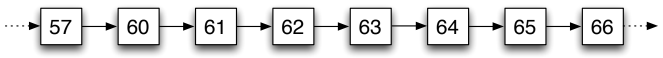


Figure 12.5: Specific system call control flow of *right-side wall follower*

Because of the common subsequences in the system call control flow of both implementations, when building up the Suffix Tree from scratch, many paths previously deleted have been reconstructed after the reconfiguration. In contrary, when using the same Suffix Tree for storing the new behavior patters, many paths already existed that matched the new behavior. The Suffix Tree was only extended by all the suffixes that include system calls requesting right-side sensors (with ID 64 and 65). After the reconfiguration, the application behavior was classified as *healthy* back again and all nodes of the currently executing paths have been marked with a *healthy/green signal*.

12.5 Discussion

In this case study, we have shown exemplarily the performance of the Online Anomaly Detection by making use of the BeBot as the application case. First, we have shown the efficiency of the IR Sensor Monitor implemented as one module of the OS Health Monitor. We distinguish between two failure models **FailureModel1** and **FailureModel4** as they both implement different subsets of Signal Generator rules. The health signals obtained for the experiments in **FailureModel1** are highly reliable in a failure-free mode as well as in case of failures. In contrary, the experiments in context of **FailureModel4** have pointed out some degree of uncertainty as the IR Sensor Monitor delivers *warning signals* in the failure-free mode and in case of failures. Furthermore, in case of a failure, the experiments have shown that *warning signals* can also recover back to a *healthy/green signal* or the failures may even be undetected lying dormant in the system.

However, based on the assumptions that failures propagate throughout the system and thereby effect the health parameters of other Component Monitor Modules, we could show the main strength of the Online Anomaly Detection: Failures can be detected by considering the context of the healths signal of the components related to the application's behavior. By this, failures can be identified even before they lead to instabilities identified by *danger signals*. The classification of behavior within a context makes the Online Anomaly Detection becoming an efficient approach that can outperform any failure detector which only examines the respective component in an isolated manner.

Part VI

Conclusion

CHAPTER 13

Summary and Conclusion

For embedded systems, dependability is a mandatory system property that becomes more challenging if these systems are applied in dynamic environments. To handle dynamically changing conditions, applications and the operating systems tend to employ autonomous principles such as self-reconfiguration, self-optimization etc. Thereby, the control of behavior is transferred to the system execution phase and requires to be evaluated in order to protect against false decisions that may harm the system's stability and, consequently, its dependability. Anomaly Detection aims at monitoring and evaluating system behavior while the system is executing, and is therefore considered as a means to enhance the dependability of a system at run-time. However, applying Anomaly Detection to self-reconfiguring real-time systems faces specific requirements which gave reason to develop a novel approach as these requirements could not be found to be fulfilled by existing Anomaly Detection Systems.

For determining the specific requirements of the approach, system characteristics of real-time systems and of self-reconfiguring systems have been examined in detail. Real-time systems, as usually being resource-restricted systems and deployed into critical application domains, need an online method to enable immediate identification of an arising threat at run-time but with low resource consumption in terms of memory usage and fully predictability in terms of computation time in order to preserve schedulability. In contrast to existing anomaly detection approaches, an anomaly detection for self-reconfiguring systems cannot rely on trained data as behavior is expected to change dynamically without any guarantee for being predictable. Because unknown behaviors may be safe as induced by a reconfiguration and known behaviors may become unstable due to reconfigurations of other system parts, it was determined that any observed behavior has to be individually classified on the basis of its threat potential by considering its effects on the underlying operating system. By excluding a training phase, a self-learning approach is required that needs history data stored in a knowledge base with an adequate coverage to enhance the accuracy of the approach.

For anomaly detection in self-reconfiguring real-time systems, behavior was defined in form of system call sequences that are executed by the application tasks. The execution of system calls has effect on the parameters of the operating system and its components that, in turn, determine the system state - defined as the Health State - and build up the context for the evaluation of the threat potential of the behavior. As the entire input data for anomaly detection, consisting of system call data and the associated health state parameter values, is kept internally by the operating system, the anomaly detection was decided to be fully integrated into the operating system, - also in order to ensure its online capability.

As an appropriate source of inspiration for the method to evaluate the behavior, the Danger Theory from Artificial Immune Systems has been identified. The idea of the Danger Theory is to classify behavior with respect to its effects on the associated environment by examining the presence or absence of danger signals. The Danger Theory enables Anomaly Detection on a context-related basis. Based on the Danger Theory, an Anomaly Detection Framework was developed that can be introduced into any real-time operating system dedicated to be applied in self-x environments. Dendritic Cells are applied to monitor application behavior on the basis of their execution of system calls. A Health State Monitor is responsible for delivering the danger signals by representing the operating system health state. Any system call execution is immediately evaluated concerning the health state resulting from its execution, and assessed whether it leads to safe, suspicious or dangerous system state. A compact data structure on the basis of Suffix Trees is applied to build up a knowledge base of history behavior. This knowledge base is established by learning all behaviors with their associated classification results. The characteristics of Suffix Trees enable online profiling of behavior and matching of behavior sequences against the conserved knowledge base in order to enhance the classification. It was shown that this approach fully satisfies all requirements and challenges formulated for anomaly detection for self-reconfiguring real-time operating systems.

The Online Anomaly Detection was implemented and integrated into the real-time operating system ORCOS which provides an execution platform for self-reconfiguring applications. The Anomaly Detection Module integrated into ORCOS mainly consists of a System Call Manager, an OS Health Monitor and a Signal Generator. In reference to the responsibility of Dendritic Cells, the System Call Monitor extracts the executed system calls that are directly included into the Suffix Tree. An OS Health Monitor was implemented composed of Component Monitor Modules assigned to each available ORCOS component for collecting their health parameters. From these health parameters, the Signal Generator obtains the system health signal by specifying a set of rules. Health signals are generated for each single component but also for representing the health state of the overall system. Because the classification accuracy mainly relies on the delivered health signal, the OS Health Monitor and the Signal Generator are fully configurable and implemented in such a manner that they can be easily extended and enhanced by expert knowledge. A discussion is provided on the memory usage caused by the Suffix Tree and the OS Health Monitor Database and it was shown that it is acceptable. As required by real-time systems, the computational overhead generated by the Online Anomaly Detection was proven to be bounded.

To evaluate the implemented Online Anomaly Detection in ORCOS, a case study was conducted by employing a virtual evaluation environment executing an autonomous BeBot application that was run on ORCOS. In this case study, the reliability of the BeBot application is dependent on the reliability of the IR sensor values that are evaluated by an IR Sensor Monitor as a specific component of the OS Health Monitor. The IR Sensor Monitor suffers from a certain degree of uncertainty in failure detection leading to unjustified warning signals. However, the experiments have shown that in case of absence of failures the unjustified warning signals affect only the IR Sensor Monitor while a true - actively injected - failure also infects other OS Health Monitor Components resulting in multiple component-related suspicious or even dangerous health states. One major result obtained by the case study is that the correlation of the health signals delivered by the distinct components enables the input health signals of Online Anomaly Detection to become more reliable. A failure that propagates throughout the system in a creeping manner can be revealed by identifying multiple components in a suspicious health state, even before a dangerous state is detected. Furthermore, in presence of a failure, the subsignals related to the specific components allow to localize the potentially failed component. As a second result, the case study has demonstrated the strength of the context-related classification: In case of a failure in an IR sensor that was detected by the IR Sensor Monitor, the application behavior operating on that IR sensor is classified as dangerous or at least as suspicious in accordance to the health signal delivered by the OS Health Monitor. That application requires a reconfiguration. In contrast, applications that do not operate on the failed IR sensor are not affected by the failure. Their behavior is classified as safe so that the application can maintain their execution.

In this thesis, a Danger Theory-based online anomaly detection approach for self-reconfiguring real-time system has been developed and evaluated. The approach allows to learn arising behaviors consisting of system call sequences executed by application tasks by storing them in a Suffix Tree-based knowledge base. The behavior being monitored is individually assessed on the basis of classification input signals reflecting the behavior's environmental context, - the operating system health state. This allows to detect unsafe behaviors independent of whether they have already occurred or they are novel ones caused by dynamical changes in the system. Reconfigurations are signaled by the Inflammation Signal to inform the Online Anomaly Detection about potential deviations in the observed execution data in order to enhance the classification and prevent false classification outcomes. The concept of the Health State Monitor consisting of distributed components responsible for evaluating the health state specific operating system components enables to detect the sources of suspicion or threats. With the correlation of the distributed component-related health signals, it becomes possible to detect failures much earlier compared to component-specific failure detection methods, especially if a potential failure tends to propagate throughout the system and its components in a creeping manner. The context-related classification enables to identify those system entities that are affected by the suspicion or threat and require a reaction, for example, by means of a reconfiguration.

The applicability of the proposed approach was discussed in detail with respect to its application domain. Furthermore, its performance in terms of effectiveness has been demonstrated. Altogether, this thesis proves the contribution to run-time dependability of self-reconfiguring real-time operating system provided by this anomaly detection approach.

However, it is important to emphasize that the accuracy of detection is strongly related to the accuracy of the classification input signals delivered by the health state evaluation. A more precise configuration of the Health State Monitor fed by expert knowledge would be required in order to make the approach more reliable. Examining each operating system component in detail, determining appropriate health parameters, configuring more accurate thresholds and specifying more adequate rules for generating signals for reflecting the operating system health state could increase the accuracy of the health state evaluation. Furthermore, the examination of the impact of the values of occurrence counter as well as the previous classification marker on a current classification outcome are still open issues and could become topics of future work.

Appendices

APPENDIX A

ORCOS System Calls

System calls supported by ORCOS can be classified into different groups of system calls. The supported system calls are:

A.1 Stream/File related system calls

System Call	ID	Arguments	Description
fopen	0	const char* filename, int blocking	Acquires the resource identified by the source path.
fclose	1	int field	Closes the resource by id and deletes the resource from the tasks owned resource database.
fread	2	void *ptr, size_t size, size_t nitems, int stream	Reads from file stream referenced by `stream` into the array pointed to by `ptr` up to `nitems` elements whose size is specified by `size` in bytes.
fwirte	3	const void *ptr, size_t size, size_t nitems, int stream	Writes from the array pointed to by `ptr` up to `nitems` elements whose size is specified by `size` to the file stream referenced by `stream`.
fputc	4	short c, int stream	Write `c` into resource `stream`.
fgetc	5	int stream	Gets the resource with id `stream` owned by this task. It may return `null` if resource not owned or not existing.
fcreate	6	const char* filename, const char* path	Creates a new resource by a `filename` at the specific `path`.

A.2 Memory related system calls

System Call	ID	Arguments	Description
new	7	size_t s	Allocates an `s` bytes of memory space for new instance.
malloc	7	size_t s	Allocates an `s` bytes of memory space for new instance.
delete	8	void* ptr	Deletes the memory space referenced by `prt`.
free	8	void *s	Deletes the memory space referenced by `s`.

A.3 Task related system calls

System Call	ID	Arguments	Description
task_stop	9	int taskid	Stops the task specified by `taskid`.
task_resume	10	int taskid	Resumes the specified taskby `taskid`. This system call effects a task that is currently not running.
getSubtaskById	68	int taskid	Gets the sub-task by its `taskid`.
changeRunningSubtask	69	int taskid	Changes the running subtask to subtask referenced by `taskid`.

A.4 Thread related system calls

System Call	ID	Arguments	Description
sleep	11	int ms	The calling thread will be blocked for at least `ms` milliseconds.
thread_create	12	int* threadid, thread_attr_t* attr, void *(*start_routine)(void*), void* arg	Creates a new thread with thread ID `threaded`, its attributes `attar` and the pointer to its start routine `start_routine`.
thread_run	13	int threadid	Starts the execution of the thread with ID defined by `threadid`.
thread_self	14		Returns the ID of the currently running thread.
thread_yield	15		Voluntarily yields the CPU to some other thread (if any other existing).
thread_exit	16		Terminates the currently executing thread.

A.5 Signal related system calls

System Call	ID	Arguments	Description
signal_wait	17	void* sig, bool memAddrAsSig	Calls a wait for the resource referenced by `sig`.
signal_signal	18	void* sig, bool memAddrAsSig	Calls a signal for the resource referenced by `sig`.

A.6 Socket related system calls

System Call	ID	Arguments	Description
socket	19	int domain, int type, int protocol, char* buffer, int buffersize	Creates a new socket with address family of type `domain`, socket type of `type`, and using the specified protocol `protocol`. For any socket an own buffer referenced by `buffer` having the size `buffer size` is specified.
connect	20	int socket, const sockaddr *toaddress	Connects to a given destination defined by `toaddress` on the provided socket by `socket`.
listen	21	int socket	Listens for connection on a socket defined by `socket`.
bind	22	int socket, const sockaddr* address	Binds a socket in `socket` to the address given by the parameter address `sockaddr`.
sendto	23	int socket, const void* buffer, size_t length, const sockaddr* dest_addr	Sends a message located the `buffer` of size `length` to the socket defined by `socket` which address is given by `dest_addr`.
recvfrom	24	int socket, char** msgptr, int flags, sockaddr* sender	Receives a message from the specified socket in `socket`. If no message is available the first thread trying to receive on this socket will be blocked. All other threads will return directly without returning a message. The blocked thread will be unblocked if a message is received by the socket and the pointer to the message is returned by the parameter `msgptr`.
add_devaddr	25	const char* dev, int domain, char* addr	Adds another `addr` to a device specified by `dev` for the domain given by `domain`.

A.7 System calls for Task loading

System Call	ID	Arguments	Description
getTasktable	33	const char* taskname, char* buffer	Requests the task table of the specified task by `taskname` and save this table into `buffer`.
create_task_physicalMemory	34	taskTable* newtaskInfo	Creates a new task under the condition of disabling virtual memory on the RAPTOR Board.
isDownloading	35		Check whether the task downloading is finished or not.
preTaskloading	36	const char* taskname, taskTable* newtask_parameters	Prepares the tasktable from `newtask_parameters` for the task defined by `task name` and prepares all necessary conditions before starting downloading the task.
createtask_physical_syscall	37	taskTable* newtask_parameters	Creates a new task with the tasktable from `newtask_parameters` under the condition of disabling virtual memory on the QEMU.

A.8 Others

System Call	**ID**	**Arguments**	**Description**
printToStdOut	26	const void* ptr, size_t max	Prints content of `ptr` with size `max` to the standard output device.
getTime	28		Gets the current system time from the timer.
map_logmemory	29	const char* log_start, const char* phy_start, size_t size, int protection	Map the desired physical address space specified by`phy_start` to the logical address space specified by `log_start`.

A.9 Specific system calls under QEMU

System Call	ID	Arguments	Description
sendto_QEMU	31	QEMUSocket *mysock, const char* data, unsigned int len, QEMUSocket *targetsock, int Protocol	Sends a message `data` of length `len` to the remote socket `targetsock` from local socket under QEMU `mysock`.
recvfrom_QEMU	32	QEMUSocket* serversock, char *buffer	Receives the message from a specific remote socket `serversock` and stores this message in a specified `buffer`.

A.10 System calls for BeBot control

System Call	ID	Arguments	Description
LEDtoRed	38		Changes all LEDs on BeBot to red.
LEDtoGreen	39		Changes all LEDs on BeBot to green.
LEDtoBlue	40		Changes all LEDs on BeBot to blue.
LEDtospecificColor	41	LEDColor* led1, LEDColor* led2, LEDColor* led3, LEDColor* led4	Changes the LEDs to specific color.
setMotorSpeed	42	unint1 leftspeed, unint1 rightspeed	Sets the speed of both motors.
stopMotor	43		Stops the BeBot motor.
readIRSensorValue	44		Gets the IR Sensor values from the BeBot.
searchWallState	45		Updates the BeBot state to search wall state.
rotateLeftState	46		Updates the BeBot state to rotate left state.
driveStraightWallState	47		Updates the BeBot state to drive straight with wall state.
rotateRightState	48		Updatse the BeBot state to rotate right state.
forceForwardState	49		Updates the BeBot state to force forward state.
leavingWallState	50		Updates the BeBot state to leaving wall state.
connectionclosed	51		Closes the connection from ORCOS to BeBot.
autodrive	52		Initiates the BeBot to autodrive in a maze game.
getBeBotPos	53		Gets the BeBot GPS value.
getTargetPos	54		Gets GPS value of the target in maze.
bonjour	55		Handshake with the remote virtual BeBot.
getIRSensorValue0	56		Gets return value of the IR Sensor 0.
getIRSensorValue1	57		Gets return value of the IR Sensor 1.
getIRSensorValue2	58		Gets return value of the IR Sensor 2.
getIRSensorValue3	59		Gets return value of the IR Sensor 3.

getIRSensorValue4	60		Gets return value of the IR Sensor 4.
getIRSensorValue5	61		Gets return value of the IR Sensor 5.
getIRSensorValue6	62		Gets return value of the IR Sensor 6.
getIRSensorValue7	63		Gets return value of the IR Sensor 7.
getIRSensorValue8	64		Gets return value of the IR Sensor 8.
getIRSensorValue9	65		Gets return value of the IR Sensor 9.
getIRSensorValue10	66		Gets return value of the IR Sensor 10.
getIRSensorValue11	67		Gets return value of the IR Sensor 11.
getCurrentState	70		Gets the current state of BeBot.
getReached	71		Checks whether the BeBot reaches the target in the maze. Returns `true` if target is reached, or `false` otherwise.
getMedianSensor	72	int index	Gets the medians of return values of the sensor referenced by `index`.
sendUniversalmsg	74	unint1 speed_left, uint1 speed_right, LEDColors* newcolor	Sends universal message to BeBot.
randomautodrive	75		Initiates the BeBot to execute the random autodrive method.
getGreenLED	82		Gets ID of green LED.
getRedLED	83		Gets ID of red LED.
getBlueLED	84		Gets ID of blue LED.

A.11 Additional System Calls for Bug Manipulator/Generator

System Call	ID	Arguments	Description
genCommBug	76		Randomized generation of manipulation data in Communication Monitor Module.
genMemBug	77		Randomized generation of manipulation data in Memory Monitor Module.
genProcBug	78		Randomized generation of manipulation data in Processor Utilization Monitor Module.
genSchedulerBug	79		Randomized generation of manipulation data in Scheduler Monitor Module.
genDevBug	80		Randomized generation of manipulation data in Device Driver Monitor Module.
genScenarioConditionBug	81		Randomized generation of manipulation data in Scenario Condition Module.

APPENDIX B

System Call Monitor API

Method Name	Description
`getMonitorMode`	Get the current monitoring mode.
`setMonitorMode`	Changes the monitoring mode to the specified one.
`recoverSystemCall`	Retrieves the system call data for the specified Thread.
`getMaxBufferCapacity`	Gets the maximum buffer capacity for the specified Thread.
`setMaxBufferCapacity`	Sets the maximum buffer capacity for the specified Thread to the specified value.
`getBufferWarning`	Gets the buffer warning size for the specified Thread.
`setBufferWarning`	Sets the buffer warning size for the specified Thread to the specified value.
`getBufferSize`	Gets the number of used entries in the buffer for the specified Thread.
`getIsBufferWarning`	Returns true if the Buffer Warning flag is set.
`getIsBufferFull`	Returns true if the Buffer Full flag is set.
`getIsBufferOverrun`	Returns true if the Buffer Overrun flag is set.

Table B.1: Monitor API: Supported Methods

APPENDIX C

OS Health Monitor - Parameter

C.1 Scheduler Monitor

Global Scheduler Monitor parameter:

Parameter	Description
`Time Interval`	The time interval from the last timer scheduler monitor execution to current time.
`Average Longterm Time Interval`	The average of time interval from system startup.
`Average Short-term Time Interval`	The average of time interval in short-term history (considering, for example, the last 10 values).

Scheduler Monitor parameter for any task J_i:

Parameter	Description
`Arrival Time`	The time at which J_i becomes ready for execution.
`Relative Deadline`	The relative deadline of the current task J_i instance.
`Absolute Deadline`	The time before which the execution of task J_i must be completed.
`Start Time`	The time at which J_i starts its execution.
`Waiting Time`	The time interval between the arrival time and the start time of current task J_i instance.
`Finishing Time`	The time at which J_i finishes its execution.
`Response Time`	The time interval between the arrival time and the finish time of current task J_i instance.
`Current Execution Time`	The time the current instance of task J_i has already taken for its execution.
`Last Execution Time`	The execution time of the last instance of task J_i.
`Average Response Time`	The average response time of all the instances of task J_i.
`Average Short-term Response Time`	The average response time of the instances of task J_i in short-term history (considering, for example, the last 10 values).
`Average Execution Time`	The average execution time of all the instances of task J_i.

`Average Short-term Execution Time`	The average execution time of the instances of task J_i in short-term history (considering, for example, the last 10 values).
`Average Waiting Time`	The average waiting time of all the instances of task J_i.
`Average Short-term Waiting Time`	The average waiting time of the instances of task J_i in short-term history (considering, for example, the last 10 values).
`Preemption Counter`	Counts the number of preemptions of the current instance of task J_i.
`Average Preemption Number`	The average number of preemptions over all the instances of task J_i.
`Average Short-term Preemption Number`	The average number of preemptions over the instances of task J_i in short-term history (considering, for example, the last 10 values).
`Preemption Time`	The time of preemption of the current instance of task J_i.
`Average Preemption Time`	The average time of preemptions over all instances of task J_i.
`Average Short-term Preemption Time`	The average time of preemptions over the instances of task J_i in short-term history (considering, for example, the last 10 values).
`Blocked Time`	The time the current instance of task J_i is blocked.
`Average Blocked Time`	The average time over all instances of task J_i are blocked.
`Average Short-term Blocked time`	The average blocked time of the instances of task J_i in short-term history (considering, for example, the last 10 values).

C.2 Processor Utilization Monitor

Global Processor Utilization Monitor parameter:

Parameter	**Description**
`Time Interval`	The time interval from the last timer of processor utilization monitor execution to current time.
`Average Longterm Time Interval`	The average of time interval from system startup.
`Average Short-term Time Interval`	The average of time interval in short-term history (considering the last 10 values).
`Idle Time`	The time in which the processor stays in idle state during the current time interval.
`Average Idle Time`	The average of idle times from system startup.
`Average Short-term Idle Time`	The average of idle times in short-term history (considering the last 10 values).
`Overall Idle Time`	The accumulated time in which the processor stays in idle state since the system startup.
`Kernel Occupying Time`	The time in the current interval that the processor is occupied by the kernel process.
`Tasks Occupying Time`	The time in the current interval in which the processor is occupied by the running periodic tasks.
`Overall Occupying Time`	The time in the current interval in which the processor is occupied by the kernel process and the tasks.
`Longterm Kernel Occupying Time`	The time in which the processor is occupied by the kernel process in longterm examination.
`Short-term Kernel Occupying Time`	The time in which the processor is occupied by the kernel process in short-term examination (considering the last 10 values).
`Longterm Tasks Occupying Time`	The accumulated time in which the processor is occupied by running periodic tasks in longterm examination.

`Short-term Task Occupying Time`	The accumulated time in which the processor is occupied by running periodic tasks in short-term examination (considering the last 10 values).
`Current Processor Utilization`	The value of the processor utilization in the current time interval.
`Longterm Processor Utilization`	The value of the longterm processor utilization calculated by the usage of the processor since the beginning of system execution.
`Short Term Processor Utilization`	The value of the short-term processor utilization is calculated by the usage of the processor within the time interval from the last time processor utilization monitor executing to current time.
`Theoretical Processor Utilization`	The value of the processor utilization determined by static analysis. Calculated offline over a long time interval, it has to be considered as a mean value to which the `Longterm Processor Utilization` shall converge.

Processor Utilization Monitor parameter for any task J_i:

Parameter	**Description**
`Time Interval`	The time interval from the last timer of processor utilization monitor execution to current time.
`Average Longterm Time Interval`	The average of time interval from system startup.
`Average Short-term Time Interval`	The average of time interval in short term history (considering the last 10 values).
`Task Occupying Time`	The time in the current interval in which the processor is occupied by that periodic task J_i.
`Overall Task Occupying Time`	The time in which the processor is occupied by that periodic task J_i since system startup.
`Short-term Task Occupying Time`	The time in which the processor is occupied by that periodic task J_i in short-term examination (considering the last 10 values).
`Current Task Utilization`	The utilization of the task within the current period.
`Average Task Utilization`	The average processor utilization of the task J_i in short-term history (considering the last 10 values).
`Overall Task Utilization`	The overall processor utilization of the task J_i since system startup.

C.3 Memory Manager Monitor

Global Memory Management Monitor parameter:

Parameter	**Description**
`Alloc Counter`	The number of calls of the `alloc()`-function (including the function calls `new()`) in the last memory monitor period.
`Alloc Size`	The allocated memory size in the last memory monitor period.
`Free Counter`	The number of calls of the `free()`-function (including the function calls `delete()`) in the last memory monitor period.
`Free Size`	The freed memory size in the last memory monitor period.

Memory Manager Monitor parameter for the kernel:

Parameter	**Description**

Memory Size	The whole memory size of the ORCOS kernel space.
Used Memory	The used memory size in the ORCOS kernel memory space.
Last Used Memory	The used memory size in the ORCOS kernel memory space at the last monitor period.
Memory Utilization	The percentage of how much memory has been used in the kernel memory space.

Memory Manager Monitor parameter for any task J_i:

Parameter	**Description**
Memory Size	The whole memory size of the specific task J_i.
Used Memory	The used memory size of the specific task J_i.
Last Used Memory	The used memory size of the specific task J_i in the last monitor period.
Memory Utilization	The percentage of how much memory has been used by the specific task J_i.
Alloc Counter	The number of calls of the `alloc()`-function (including the function calls `new()`) in the last memory monitor period by the task J_i.
Alloc Size	The allocated memory size by the task J_i in the last memory monitor period.
Free Counter	The number of calls of the `free()`-function (including the function calls `delete()`) in the last memory monitor period by the task J_i.
Free Size	The freed memory size by the task J_i in the last memory monitor period.

C.4 Communication Monitor

Global Communication Monitor parameter:

Parameter	**Description**
Time Interval	The time interval from the last timer of communication monitor execution to current time.
Average Longterm Time Interval	The average of time interval from system startup.
Average Short-term Time Interval	The average of time interval in short term history (considering the last 10 values).
Upload	The quantity of uploaded content in this time interval.
Download	The quantity of downloaded content in this time interval.
CurrentWorkload	The quantity of all communication content in this time interval.
Upload Speed	The speed of uploading in this time interval.
Download Speed	The speed of downloading in this time interval.
Communication Speed	The speed of uploading and downloading in this time interval.
Average Communication Speed	The average speed of uploading and downloading.
Communication Load	The quantity of all communication content since the system executing.
cOk	The number of `cOk` the communication module got as the return value in this time interval.
cError	The number of `cError` (communication failures) the communication module got as the return value in this time interval.

`Overall cOk`	The number of `cOk` the communication module got as the return value since system startup.
`Average cError`	The average number of `cError` (communication failures) the communication module got as the return value.
`Overall cError`	The number of `cError` (communication failures) the communication module got as return value since system startup.

Communication Monitor parameter for any task J_i:

Parameter	**Description**
`Time Interval`	The time interval from the last timer of communication monitor execution to current time.
`Average Longterm Time Interval`	The average of time interval from system startup.
`Average Short-term Time Interval`	The average of time interval in short term history (considering the last 10 values).
`Upload`	The quantity of uploaded content in this time interval for the task J_i.
`Download`	The quantity of downloaded content in this time interval for the task J_i.
`CurrentWorkload`	The quantity of all communication content in this time interval for the task J_i.
`cOk`	The number of times of communication module gets `cOk` as the returned value in that period for the task J_i.
`cError`	The number of times of communication module gets `cError` as the returned value in this time interval for the task J_i. .
`Overall cOk`	The number of times of communication module gets `cOk` as the returned value for the task J_i.
`Overall cError`	The number of times of communication module gets `cError` as the returned value for the task J_i.
`Average cError`	The average times of communication module gets `cError` as the returned value per period for the task J_i.

C.5 File Manager Monitor

Global File Manager Monitor parameter:

Parameter	**Description**
`cOk`	The number of times the file manager returns `cOK` when processing file-related functions in the current time interval.
`cError`	The number of times the file manager returns `cError` processing file-related functions in the current time interval.
`Last cError`	The number of times the file manager returns `cError` in the last monitor period.
`Overall cError`	The overall number of times the file manager returns `cError` when processing file-related functions.
`cResourceNotOwnedTimes`	The number of times the file manager returns `cResourceNotOwnedTimes` in the current time interval.

`Last cResourceNotOwnedTimes`	The number of times the file manager returns `cResourceNotOwnedTimes` in the last monitor period.
`Overall cResourceNotOwnedTimes`	The overall number of times the file manager returns `cResourceNotOwnedTimes` when processing file-related functions.
`cResourceNotWriteableTimes`	The number of times the file manager returns `cResourceNotWriteableTimes` in the current time interval.
`Last cResourceNotWriteableTimes`	The number of times the file manager returns `cResourceNotWriteableTimes` in the last monitor period.
`Overall cResourceNotWriteableTimes`	The overall number of times the file manager returns `cResourceNotWriteableTimes` when processing file-related functions.
`cResourceNotReadableTimes`	The number of times the file manager returns `cResourceNotReadableTimes` in the current time interval.
`Last cResourceNotReadableTimes`	The number of times the file manager returns `cResourceNotReadableTimes` in the last monitor period.
`Overall cResourceNotReadableTimes`	The overall number of times the file manager returns `cResourceNotReadableTimes` when processing file-related functions.
`cInvalidResourceTimes`	The number of times the file manager returns `cInvalidResourceTimes` in the current time interval.
`Last cInvalidResourceTimes`	The number of times the file manager returns `cInvalidResourceTimes` in the last monitor period.
`Overall cInvalidResourceTimes`	The overall number of times the file manager returns `cInvalidResourceTimes` when processing file-related functions.

File Manager Monitor parameter for any task J_i:

Parameter	**Description**
`cOk`	The overall number of times the file manager returns `cOK` when the task J_i is processing file-related functions.
`cError`	The number of times the file manager returns `cError` for the task J_i in this time interval.
`Last cError`	The number of times the file manager returns `cError` for the task J_i in the last monitor period.
`Overall cError`	The overall number of times the file manager returns `cError` when the task J_i processing file-related functions.
`cResourceNotOwnedTimes`	The number of times the file manager returns `cResourceNotOwnedTimes` for the task J_i in this time interval.
`Last cResourceNotOwnedTimes`	The number of times the file manager returns `cResourceNotOwnedTimes` for the task J_i tin the last monitor period.

`Overall cResourceNotOwnedTimes`	The overall number of times the file manager returns `cResourceNotOwnedTimes` when the task J_i processing file-related functions.
`cResourceNotWriteableTimes`	The number of times the file manager returns `cResourceNotWriteableTimes` for the task J_i in this time interval.
`Last cResourceNotWriteableTimes`	The number of times the file manager returns `cResourceNotWriteableTimes` for the task J_i tin the last monitor period.
`Overall cResourceNotWriteableTimes`	The overall number of times the file manager returns `cResourceNotWriteableTimes` when the task J_i processing file-related functions.
`cResourceNotReadableTimes`	The number of times the file manager returns `cResourceNotReadableTimes` for the task J_i in this time interval.
`Last cResourceNotReadableTimes`	The number of times the file manager returns `cResourceNotReadableTimes` for the task J_i tin the last monitor period.
`Overall cResourceNotReadableTimes`	The overall number of times the file manager returns `cResourceNotReadableTimes` when the task J_i processing file-related functions.
`cInvalidResourceTimes`	The number of times the file manager returns `cInvalidResourceTimes` for the task J_i in this time interval.
`Last cInvalidResourceTimes`	The number of times the file manager returns `cInvalidResourceTimes` for the task J_i tin the last monitor period.
`Overall cInvalidResourceTimes`	The overall number of times the file manager returns `cInvalidResourceTimes` when the task J_i processing file-related functions.
`numberofResource`	The number of resources that belong to this task.
`Last numberofResource`	The number of resources that have belong to this task in the last monitor period.

C.6 Device Driver Monitor

Global Device Driver Monitor parameter:

Parameter	**Description**
`Time Interval`	The time interval from the last timer device driver monitor execution to current time.
`Average Longterm Time Interval`	The average of time interval from system startup.
`Average Short-term Time Interval`	The average of time interval in short-term history (considering, for example, the last 10 values).

C.7 IR Sensor Monitor

Global IRSensor Monitor parameter:

Parameter	**Description**
`Time Interval`	The time interval from the last timer IR sensor monitor execution to current time.
`Average Longterm Time Interval`	The average of time interval from system startup.
`Average Short-term Time Interval`	The average of time interval in short-term history (considering, for example, the last 10 values).
`Sensor Value` *i*	The current value of the sensor with index *i*. The IR Sensor Monitor holds one data field per each IR Sensor *i*.
`Last Sensor Value` *i*	The value of the sensor with index *i* of the last period. The IR Sensor Monitor holds one data field per each IR Sensor *i*.

APPENDIX D

Device Driver Monitor Interface

Method	Description
`UpdateData`	Update the collected data and collecting time interval.
`UpdateSignall`	Update the health signal of the device driver and report it to the OS Health Monitor Center.
`checkstatus`	Responsible for the implementation of data collection and evaluation (relations and conditions).

APPENDIX E

OS Health Monitor API

Method	Description
`getHealthPool`	Get the pointer of the first health matrix in the health pool.
`getLastHealthMatrix`	Get the last updated health matrix from the health pool.
`getHealthMatrixbyIndex`	Get the health matrix through inputted index.
`triggerAnomalyDetection`	An API left to the future anomaly detection. To trigger the anomaly detection module to double-analyze the history health data.
`updateData`	To collect the health matrix from all enabled health monitor modules.
`getCommunicationMonitor`	Get the pointer of the communication monitor module if the communication monitor is enabled.
`getMemoryMonitor`	Get the pointer of the memory monitor module if the memory monitor is enabled.
`getIRSensorMonitor`	Get the pointer of the IR sensor monitor module if the IR sensor monitor is enabled.
`getProcessorUtilizationMonitor`	Get the pointer of the processor utilization monitor module if the processor utilization monitor is enabled.
`getSchedulerMonitor`	Get the pointer of the scheduler monitor module if the scheduler monitor is enabled.
`getScenarioConditionMonitor`	Get the pointer of the scenario condition monitor module if the scenario condition monitor is enabled..

APPENDIX F

Proposals for future research

F.1 Potentials to enhance the evaluation of the classification marker

F.1.1 Previous Classification Marker Value

The previous classification marker value of a node is the value of the node's classification marker set by an instance executed before the current task instance execution. Whenever a node in the SuffixTree is reached by the current behavior sequence, first, the value of the classification marker is transferred to the node's `previous_classification_marker` (as a further attribute of the class `TreeNode`) before the evaluation of the health signal related to current execution is performed. In fact, the previous classification marker value represents the health state referring to the latest execution of a task instance whose behavior sequence was completely matching the current behavior sequence, - at least until the symbol represented by the currently considered node.

Having information about the health signal related to, at least partially, equal behavior sequences may be of interest for examining the health state of the current instance. Therefore, as a first indicator, we propose to take into account potential effects on the current value of a node's classification marker provided by the node's previous classification marker value.

Different scenarios have been identified in which it could be possible to gain information based on the previous classification marker value:

1. Combining the obtained classification result with the previous classification marker value to obtain the current classification marker value.

   Examining classification results of equal (sub-) sequences enables to adduce evidence of the general effects caused by that pattern.

   On the one hand, it is possible, that the execution of a behavior pattern always leads to the same classification result. If this result is a *safe/green* signal, then the task's behavior is in best order. But if the classification of a behavior sequence results in a *warning* or even *danger* signal, then the task might contain a general problem in its behavior. Obviously, this cannot be detected if only one (the current one) classification result is considered.

   On the other hand, it is also possible that the accumulations of equal behavior patterns reinforce values of health parameters through creeping effects on the system health state provoked by that patterns. A pattern which execution initially yield to a *safe* signal, may evolve to a *warning* signal after a certain number of executions, and may lead to a *danger* signal after another number of executions of that pattern (for example by increasing the `Memory Usage` by each execution of the behavior sequence). To be able to assess the classification results of the current execution, it is essential to take into account the previous classification results of formerly occurring task instances.

In case of existing general instability in a task's behavior, we deal in most of the cases with persistent problems. Therefore, it can be assumed that the resulting health signals associated with behavior patterns will not be able to recover when reaching a state of being classified as suspicious or dangerous. Moreover the instability may propagate throughout the entire system and - if classified as *yellow* first - will potentiallly lead into a dangerous state after a certain number of executions. Based on this assumptions, we can formulate ideas how to integrate the value of the `previous_classification_marker` into the computation of the current classification marker.

- If the current node's classification result is *green/healthy*, the value of the node's previous classification marker has no impact on the current value. Even if the value of the previous classification marker was *yellow/warning* or *red/danger*, then a subsequent *green* health signal is an evidence for a recovery of the system's health state. Such a situation may be an indicator that the former classification might have been caused by a transient execution inconsistency without any consequences.
- If the current node's classification result is (at least) *yellow/warning*, then the value of the node's previous classification marker will be evaluated. If the value of the previous classification marker is also *yellow/warning*, then the probability of a persistent inconsistency is identified. To prevent a potential creeping of an inconsistency, subsequent *yellow/warning* signals associated with one behavior pattern can be used either to notify the responsible controller or to enforce the obtained health signal to change into *red/danger* signal and thereby provoke the controller to induce a reaction at an early stage.
- In contrast to the former simple rules, a more detailed specification of the impact of the previous classification marker value is possible. Therefore, a useful means is to introduce different degrees of values for danger, warning or safe signals. Adding further attributes such as *light, medium, high* to the values of the classification marker could enable the evaluate the occurrence of subsequently equal health signals. Different concepts are possible here.

  One possible approach to integrate the previous classification marker value is illustrated on a classification outcome of a *yellow/warning* signal: A first occurrence of a *yellow/warning* signal may be set by the attribute *light*, while the subsequent occurrence of the *yellow/warning* signal will reinforce the attribute to *medium*, and to *high* if the classification marker keeps to maintain the *yellow/warning* signal value for a consecutive execution. One proposition is based on the assumption that the value of the classification outcome is maintained in the classification marker value with the previous classification marker value having only impact on the attribute is illustrated in Figure F.1.

  Another option also based on the assumption that the value of the classification outcome is maintained in the classification marker but including the exception in case of subsequently occurring *yellow/warning* signals that will - after a particular rate - lead to a *red/danger* signal as illustrated in the matrix provided by Figure F.2.

  Besides these two options proposed here, many other assignment strategies for the impact of the previous classification marker are feasible also (such as step-wise manipulation), but they will not be discussed in further details here.

  The most challenging question is how to assign a value to the current classification marker if it differs from the previous classification marker value, especially how to deal with *red* and *green* signals. Is a *green* signal following a previously obtained *yellow* signal with the attribute *high* a real recovery leading to a *green/health* value of the current classification marker (with which attribute value?) or shall it step-wise decrease the *yellow* signal to having the attribute *medium*? Could such reinforcement of the classification marker value lead to potential false alarms? These questions can only be answered by precise analysis and specification of an anomaly detection strategy.
- A last (but not least) idea is to extend each node by a counter of signal occurrence. In practice, this can be implemented by using the example of the general node's occurrence

previous classification marker value / **current classification outcome**	green/ healthy light	green/ healthy medium	green/ healthy high	yellow/ warning light	yellow/ warning medium	yellow/ warning high	red/danger light	red/danger medium	red/danger high
green/healthy	green/ healthy light	green/ healthy light	green/ healthy light	green/ healthy medium	green/ healthy medium	green/ healthy medium	green/ healthy high	green/ healthy high	green/ healthy high
yellow/warning	yellow/ warning light	yellow/ warning light	yellow/ warning light	yellow/ warning medium	yellow/ warning high	yellow/ warning high	yellow/ warning high	yellow/ warning high	yellow/ warning high
red/danger	red/danger light	red/danger light	red/danger light	red/danger light	red/danger medium	red/danger medium	red/danger high	red/danger high	red/danger high

Figure F.1: Impact of previous classification marker on value of current classification marker.

previous classification marker value / **current classification outcome**	green/ healthy light	green/ healthy medium	green/ healthy high	yellow/ warning light	yellow/ warning medium	yellow/ warning high	red/danger light	red/danger medium	red/danger high
green/healthy	green/ healthy light	green/ healthy light	green/ healthy light	green/ healthy medium	green/ healthy medium	green/ healthy medium	green/ healthy high	green/ healthy high	green/ healthy high
yellow/warning	yellow/ warning light	yellow/ warning light	yellow/ warning light	yellow/ warning medium	yellow/ warning high	**red/ danger light**	yellow/ warning high	yellow/ warning high	yellow/ warning high
red/danger	red/danger light	red/danger light	red/danger light	red/danger light	red/danger medium	red/danger medium	red/danger high	red/danger high	red/danger high

Figure F.2: Accumulated impact of previous classification marker on value of current classification marker.

counter. After getting a particular health signal for a node, the node's appropriate signal occurrence counter is incremented.

Two different approaches are possible to manage the signal occurrence counter:

1. to count every occurrence of a health signal in order to evaluate its relation to the overall node's occurrence
2. to count consecutive occurrences of a health signal in order to detect a sustained health state

For both approaches, thresholds can be determined to enhance the value of the classification outcome in order to set the classification marker value such as: raise the classification marker value to *red/danger* afternthe *i*th consecutive occurrence of a *yellow/warning* health signal.

By processing the classification method, a result in form of a health signal is achieved. This classification result can be adapted by impact of the node's previous classification marker. The several approaches presented here can be applied to incorporate the value of the previous classification marker with each having its different qualities and complexities to be evaluated.

2. Parameterizing the classification method

The formerly discussed ideas rely on the preliminarily processing of the classification method as originally defined. Another possibility to incorporate the previous classification marker directly into the classification method.

The classification method is based on using Signal Generator rules that refer to thresholds, define ranges and decision or parameter weights to generate a health signal. However, processing the effected Signal Generator rules may lead to nondistinctive situations such as

- classification results to be close to boundaries between two classification classes

- classification results in between signal boundary ranges
- unclear classification outcome because of different sub-signals obtained by different rules that operate of exclusive subsets of health parameters

For example, the classification of a node results in a *yellow/warning* signal but with parameter values that are close to the boundary of being classified as dangerous. Knowing that the previous classification marker value is for example *yellow/warning* could tighten the decision making rules. The value of previous classification marker can take the role of providing a tendency of the health state in order to immediately raise a signal (e.g. *red/danger* signal) anticipated by further executions. This can only be implemented by adapting the Signal Generator rules and integrating a further parameter in forms of the previous classification marker.

Various effects of the previous classification marker value are possible on the Signal Generator rules:

- as a function of the value of the previous classification marker strengthening particular health parameters to reinforce the classification outcome while weakening other health parameters
- weighting particular Signal Generator rules
- balancing the weights of the rules belonging to the effected subset of Signal Generator rules

The adaption of the Signal Generator rules, obviously, requires further analysis of the semantics of health parameters and of the potentials of the previous classification marker value to effect the current classification decision.

Information about the previous classification marker value can be exploit to enhance the evaluation and improve the precision of the obtained health signal. By incorporating the previous classification marker into the evaluation of the current classification marker, the classification outcome can become more sensitive and may assess an anticipated system's health state at an early stage.

F.1.2 Occurrence Counter

Based on the nature of Suffix Trees, the number of occurrences of sequences can be recorded in the leaf nodes whenever the leaf node is reached by the executing behavior sequence. Furthermore, the number of occurrence of unique and coherently executed subsequences that are outlined being located in between two branch nodes can also be derived based on the characteristics of Suffix Trees. On the one hand, it is possible to record the number of occurrences of a branch node individually such as implemented for the leaf node. On the other hand, it can be calculated based on the sum of number of occurrence of those leaf nodes that are descending from the branch node currently considered.

Besides the previous classification marker value, the Occurrence Counter is another factor that may contribute to the classification method. It provides information whether a particular behavior pattern is of high or low occurrence. Furthermore, the frequency of a pattern can be derived from the occurrence counter with respect to the entire system lifecycle as well as with respect to a short-term history.

High occurrence of a behavior pattern identifies a common behavior. In this context, some health parameters such as for example those that are associated to provide average values, are expected to be *stable*. One approach to incorporate the Occurrence Counter into the classification method, in case of a high occurrence, is to strengthen the weights of those health parameters that are stable for common behavior.

Different reasons lead to behavior patterns of low occurrence: Initialization is mostly executed at the startup of a system or an application only and results in a behavior pattern that is specific for initialization phase. Seldomly occurring conditions or correlations may lead to a deviation of behavior patterns. New behavior (for example, after a system reconfiguration) will lead to patterns not known before, having zero or low occurrence initially. But also unstable behavior leading to a potential failure is expected to be of low occurrence. Behavior patterns of low occurrence, of course, also have to be classified according to the defined rules. However, for behaviors of low occurrence, the health parameters, especially those being stable for common behaviors such as global parameters or average values, are expected to deviate. Furthermore, health parameters can be identified that might be of more relevance in anomalous behavior patterns in order to identify potentially critical situations. These health parameters and rules operating on these health parameters should be given according weights to enhance the precision of the classification method.

As a further add-on, by considering the occurrence counter and the resultant frequency counter, mechanisms of learning could be integrated into the Signal Generator rules where weights and parameters evolve with respect to the occurrences of behavior patterns.

F.2 Alternatives for Processing the Anomaly Detection

Wse cases are conceivable in which anomaly detection is required but to be performed in an online manner because of diverse reasons: This could be related to performance or resource restrictions, to priority policy and assignment of priorities to other components and tasks or related to a lower demand in the detection precision. No matter what the reason is to decrease the execution priority of the Anomaly Detection Framework, different stages are determined to alter and interfere the anomaly detection workflow aligned with a system specification:

2. A first approach to reduce the runtime overhead of the anomaly detection is to abandon the strict procedure of classifying every system call. Instead, system calls are classified selectively according to different strategies:

   2.1 maintain the classification of system calls that are represented as branch nodes and leaf nodes.

   2.2 maintain the classification only of leaf nodes as representatives of the behavior sequences associated with task instances

   2.3 maintain the classification only of specific system calls (either belonging to a set of system calls or even individually selected system calls)

   Each node skipped in terms of classification reduces the runtime costs by the effort produced by the classification method. Nevertheless, to ensure the determinism of the anomaly detection (and consequently the entire system), a pessimistic worst case analysis is required.

   This is simple for the second and the third case: For the second case, the classification is processed at the end of each behavior sequence. The resulting classification costs for each task instance are determined by one execution of the classification method (instead of executing the classification method for n-times, with n being the length of the behavior sequence). Considering the third case of classifying only dedicated system calls, the classification cost are determined by the maximum number of occurrence of the dedicated system calls within a behavior sequence. If a behavior sequence consists only of those system calls that belong to the set of system calls to be classified, no real reduction of the runtime overhead can be achieved.

   The discussion of reducing the runtime cost for classification becomes more complicated for the first case: Again in worst case, every system call in a behavior sequences can take on a branch node. The more interesting question is whether it is possible to verify in advance all the system calls that in the Suffix Tree will be represented by a branch node? To realize this, all behavior patterns must to know in advance which actually contradicts with the assumption set on this thesis to classify behaviors being previously unknown. A clear assessment of the runtime effort for classification - if only reduced to branch and leaf nodes - is therefor not possible.

3. In order to preserve fully determinism and schedulability, the Anomaly Detection Module can be scheduled as an ordinary periodic worker thread (kernel thread) with a dedicated priority. Classification is then not directly related to a particular system call or a sequence but to the current system time stamp.

4. In order to not interfere in the system performance, and to not interfere with the running tasks and the present RTOS components, anomaly detection and classification can be shifted to only idle times of the system. This approach is actually associated with background scheduling.

5. Considering the latter strategies to reduce runtime overhead, they are all concentrating on reducing the cost produced by the classification. A further option is to address the cost related to the Knowledge Base. The extension of the Suffix Tree by current system calls contributes to the runtime overhead of the Anomaly Detection Framework. The system calls are recorded by the System Call Monitor which additionally offers data structures to hold the system call information. Hence, the these information do not need to be immediately included into the Knowledge Base if the classification of them is also postponed as proposed by one of the last two strategies (priority-based execution or backgound scheduling).

   Nevertheless, the costs that remain in any case are the ones related to system call monitoring. If anomaly detection is applied as presented in this theses on the basis of behavior patterns, the part of the System Call Monitor is undisputed and cannot be suspended in the workflow of system call execution.

By choosing any of the strategies to reduce runtime overhead, contemporary, the detection precision is degraded. Generally, each skipping of a node leads to losing the information that could have been potentially obtained by classifying the node. Effected health parameters may be overwritten by following system calls.

Moreover, in periodic or background scheduling of the anomaly detection, the classification performed is not directly associated with a specific behavior entity, but related to the application behavior proceeded since the last classification. Then, it becomes impossible to allocate the definite source of problem if a problem (in forms of a *yellow/warning* or even *red/danger* signal) will be identified. The context of the classification is loosed.

Hence, reducing the runtime cost of classification is bound on information loss.

6. A last option is to apply hybrid approaches:

   6.1 For known behavior sequences that are already recorded in the Knowledge Base, the classification is performed at dedicated classification entities, e.g. classifying the behavior only at leaf nodes. If a novel behavior pattern occurs, it obviously initiates an extension of the Knowledge Base by adding new nodes into the Suffix Tree. When registering new behavior patterns, the classification is processed online for each new node in the behavior pattern. By this, a detailed and thorough classification of novel behavior patterns is ensured in order to directly detect potential threats resulting from this previously unknown behavior.

   6.2 Initially, the classification is performed according to one of the overhead-reducing strategies presented above. This is maintained as long as the system health state is *green/healthy*. However, if any deviation of the health state is detected, it is necessary to induce more precision into the classification. In such a case, the scheduling of the classification is reconfigured and set to the optimal classification flow. With this adaptation, the classification is set to work online again which allows immediate classification and detection of the source of potential threats. Although, the adaptation of the classification precision can only take effect at the earliest executed for the next task instance.

   Running the anomaly detection in this hybrid approach, however, requires to guarantee the resources for worst-case - which is defined by the runtime overhead of the optimal flow - at any point of time. In order to implement this, the Profile Framework (see Section 8.1.1) is exploit: different profiles are defined for the different classification granularities specified by the different strategies, but with a guarantee of being able to reconfigure to the worst-case profile - the optimal flow - and schedule it. While running the anomaly detection in a higher quality profile (with lower runtime overhead), the resources that stay unused can be made available for the purposes of enhancing performance of tasks or system components. In worst case, they become available for the anomaly detection purpose.

After choosing a strategy, appropriate execution periods and priorities, all parameters can be configured offline in the SCL Configuration referring to the Anomaly Detection Module. It must be emphasized, that even if included into this concept, the hybrid strategy is not fully implemented yet.

All in all, the system designer must be aware of the required detection precision, if the classification on demand is not realizable because of unacceptable classification costs. Furthermore, the system designed must also be aware of the risk to lose information as the context-defining health parameters are volatile with respect to a continuous system execution.

List of Figures

My Publications

[Dellnitz et al., 2014] Dellnitz, M., Flasskamp, K., Hartmann, P., Krüger, M., Meyer, T., Priesterjahn, C., Ober-Blöbaum, S., Rasche, C., Sextro, W., Stahl, K., and Trächtler, A. (2014). Self-optimizing mechatronic systems. In Gausemeier, J., Rammig, F.-J., Schäfer, W., and Sextro, W., editors, *Dependability of Self-optimizing Mechatronic Systems*, chapter 1.1, pages 3–12. Springer-Verlag, Heidelberg, Germany.

[Stahl, 2013] Stahl, K. (2013). Ais-based anomaly detection for self-x systems. In *Proceedings of the First Organic Computing Doctoral Dissertation Colloquium (OC-DDC'13)*, pages 24 – 26.

[Stahl et al., 2014a] Stahl, K., Groesbrink, S., and Oberthür, S. (2014a). System software. In Gausemeier, J., Schäfer, W., and Rammig, F.-J., editors, *Design Methodology for Intelligent Technical Systems - Develop Intelligent Technical Systems of the Future*, chapter Methods for the Design and Development, pages 298–317. Springer-Verlag, Heidelberg, Germany.

[Stahl and Rammig, 2014] Stahl, K. and Rammig, F.-J. (2014). Online behavior classification for anomaly detection in self-x real-time systems. In *Proc. 5th IEEE Workshop on Self-Organizing Real-Time Systems (SORT) 2014*. IEEE, IEEE.

[Stahl and Rammig, 2015] Stahl, K. and Rammig, F.-J. (2015). Online behavior classification for anomaly detection in self-x real-time systems. *Concurrency and Computation: Practice and Experience*.

[Stahl et al., 2013] Stahl, K., Rammig, F.-J., and Vaz, G. (2013). A framework for enhancing dependability in self-x systems by artificial immune systems. In *Proc. 4th IEEE Workshop on Self-Organizing Real-Time Systems (SORT) 2013*. IEEE, IEEE.

[Stahl et al., 2014b] Stahl, K., Seifried, A., Trächtler, A., Kleinjohann, B., Korf, S., Porrmann, M., Heinzemann, C., Rasche, C., Sondermann-Woelke, C., Priesterjahn, C., Steenken, D., Rammig, F.-J., Wehrheim, H., Kessler, J. H., Gausemeier, J., Flasskamp, K., Witting, K., Kleinjohann, L., Krüger, M., Dellnitz, M., Iwanek, P., Reinold, P., Hartmann, P., Dorociak, R., Timmermann, R., Ober-Blöbaum, S., Groesbrink, S., Ziegert, S., Xie, T., Meyer, T., Sextro, W., Schäfer, W., Müller, W., and Zhao, Y. (2014b). Methods of improving the dependability of self-optimizing systems. Dependability of Self-Optimizing Mechatronic Systems, pages 37–171. Springer Verlag.

[Stahl et al., 2015] Stahl, K., Stöcklein, J., and Li, S. (2015). Evaluation of autonomous approaches using virtual environments. In *Virtual, Augmented and Mixed Reality - 7th International Conference, VAMR 2015, Held as Part of HCI International 2015, Los Angeles, CA, USA, August 2-7, 2015, Proceedings*, pages 499–512.

Bibliography

[1] Heinz Nixdorf Institute, The BeBot mini robot. https://www.hni.uni-paderborn.de/en/system-and-circuit-technology/projects/abgeschlossene-projekte/mini-robot-bebot/, Aug. 2015.

[2] ORCOS - Organic Reconfigurable Operating System, FG Rammig, University of Paderborn. https://orcos.cs.uni-paderborn.de/doxygen/html/.

[3] Organic Computing Initiative, DFG Priority Program 1183. http://www.organic-computing.de.

[4] QEMU - open source processor emulator. http://www.qemu.org, Jul. 2015.

[5] Wikipedia: ARIMA (and ARMA). http://en.wikipedia.org/wiki/Autoregressive_integrated_moving_average, Dec. 2014.

[6] Wikipedia: Artificial Neural Networks. http://en.wikipedia.org/wiki/Artificial_neural_network, Dec. 2014.

[7] Wikipedia: Bayesian Networks. http://en.wikipedia.org/wiki/Bayesian_network, Dec. 2014.

[8] Wikipedia: Clustering, Cluster Analysis. http://en.wikipedia.org/wiki/Cluster_analysis, Dec. 2014.

[9] Wikipedia: Extended Finite State Automata. http://en.wikipedia.org/wiki/Extended_finite-state_machine, Dec. 2014.

[10] Wikipedia: Hidden Markov Model. http://en.wikipedia.org/wiki/Hidden_Markov_model, Dec. 2014.

[11] Wikipedia: K-means Clustering. http://en.wikipedia.org/wiki/K-means_clustering, Dec. 2014.

[12] Wikipedia: Kernel Methods, Kernel Trick. http://en.wikipedia.org/wiki/Kernel_trick, Dec. 2014.

[13] Wikipedia: Local Outlier Factor. http://en.wikipedia.org/wiki/Local_outlier_factor, Dec. 2014.

[14] Wikipedia: Regression Analysis. http://en.wikipedia.org/wiki/Regression_analysis, Dec. 2014.

[15] Wikipedia: Self-Organizing Maps. http://en.wikipedia.org/wiki/Self-organizing_map, Dec. 2014.

[16] Wikipedia: Statistical Hyphothesis Testing. http://en.wikipedia.org/wiki/Statistical_hypo thesis_testing, Dec. 2014.

[17] Wikipedia: Support Vector Machines. http://en.wikipedia.org/wiki/Support_vector_machine, Dec. 2014.

[18] Wikipedia: Virtual Reality. https://en.wikipedia.org/wiki/Virtual_reality, Aug. 2015.

[19] U. Aickelin, P. Bentley, S. Cayzer, J. Kim, and J. McLeod. Danger theory: The link between ais and ids? In J. Timmis, P. Bentley, and E. Hart, editors, *Artificial Immune Systems*, volume 2787 of *Lecture Notes in Computer Science*, pages 147–155. Springer Berlin Heidelberg, 2003.

[20] U. Aickelin and S. Cayzer. The danger theory and its application to artificial immune systems. In J. Timmis and P. J. Bentley, editors, *Proceedings of the 1st International Conference on Artificial Immune Systems (ICARIS-2002)*, pages 141–148, University of Kent at Canterbury, September 2002. University of Kent at Canterbury Printing Unit.

[21] J. Al-Enezi, M. Abbod, and S. Alsharhan. Artificial immune systems - models, algorithms and applications. *International Journal of Research and Reviews in Applied Sciences (IJRRAS)*, 3(3):118–131, 2010).

[22] S. Axelsson. Research in intrusion-detection systems: A survey. Technical Report 98–17, Department of Computer Engineering, Chalmers University of Technology, SE–412 96, Göteborg, Sweden, Dec. 1998.

[23] S. Axelsson. Intrusion detection systems: A survey and taxonomy. Technical report, 2000.

[24] O. Babaoglu, M. Jelasity, A. Montresor, C. Fetzer, S. Leonardi, A. van Moorsel, and M. van Steen. The self-star vision. In O. Babaoglu, M. Jelasity, A. Montresor, C. Fetzer, S. Leonardi, A. van Moorsel, and M. van Steen, editors, *Self-star Properties in Complex Information Systems*, volume 3460 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin Heidelberg, 2005.

[25] B. B. Bayer. Reflexive selbstheilende Komponenten zur Unterstützung für ein eingebetteten Echtzeit-Betriebssystem. Master thesis, Faculty of Computer Science, Electrical Engineering, and Mathematics, Paderborn University, 2010.

[26] D. J. Berndt and J. Clifford. Using Dynamic Time Warping to Find Patterns in Time Series. In *KDD Workshop*, pages 359–370, 1994.

[27] D. J. Brown, B. Suckow, and T. Wang. A survey of intrusion detection systems. Technical report, Department of Computer Science, University of California, 2002.

[28] M. Burgess. Probabilistic anomaly detection in distributed computer networks. *Science of Computer Programming*, page 2006.

[29] M. Burgess. Recent developments in cfengine. In *In Proceedings of the 2nd Unix.nl conference*, 2001.

[30] G. C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series)*. Springer-Verlag TELOS, Santa Clara, CA, USA, 2004.

[31] L. N. d. Castro. *Fundamentals of Natural Computing (Chapman & Hall/Crc Computer and Information Sciences)*. Chapman & Hall/CRC, 2006.

[32] L. R. d. Castro and J. Timmis. *Artificial Immune Systems: A New Computational Intelligence Paradigm*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.

[33] V. Chandola. *Anomaly detection for symbolic sequences and time series data*. Phd thesis, University of Minnesota, 2009.

[34] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3):15:1–15:58, July 2009.

[35] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection for discrete sequences: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 24(5):823–839, 2012.

[36] V. Chandola, V. Mithal, and V. Kumar. Comparative evaluation of anomaly detection techniques for sequence data. In *Proceedings of the 2008 Eighth IEEE International Conference on Data Mining*, ICDM '08, pages 743–748, Washington, DC, USA, 2008. IEEE Computer Society.

[37] D. Dasgupta, S. Yu, and F. Nino. Review article: Recent advances in artificial immune systems: Models and applications. *Appl. Soft Comput.*, 11(2):1574–1587, Mar. 2011.

[38] L. de Castro. The immune response of an artificial immune network (ainet). In *Evolutionary Computation, 2003. CEC '03. The 2003 Congress on*, volume 1, pages 146–153 Vol.1, Dec 2003.

[39] L. de Castro and F. Von Zuben. Learning and optimization using the clonal selection principle. *Evolutionary Computation, IEEE Transactions on*, 6(3):239–251, Jun 2002.

[40] A. V. Debra Anderson, Thane Frivold. Next-generation intrusion detection expert system (nides) - a summary. Technical Report SRI-CSL-95-07, SRI International, Menlo Park, CA 94025-3493, May 1995. This report was prepared for the Department of the Navy, Space and Naval Warfare Systems Command, under Contract N00039-92-C-0015.

[41] S. Even. *Graph Algorithms*. W. H. Freeman & Co., New York, NY, USA, 1979.

[42] T. Fawcett and F. Provost. Activity monitoring: Noticing interesting changes in behavior. In *In Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 53–62, 1999.

[43] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *In Proceedings of the 2003 IEEE Symposium on Security and Privacy*, 2003.

[44] S. Forrest, S. Hofmeyr, and A. Somayaji. The evolution of system-call monitoring. In *Proceedings of the 2008 Annual Computer Security Applications Conference*, ACSAC '08, pages 418–430, Washington, DC, USA, 2008. IEEE Computer Society.

[45] S. Forrest, S. A. Hofmeyr, and A. Somayaji. Computer immunology. *Commun. ACM*, 40(10):88–96, Oct. 1997.

[46] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, SP '96, pages 120–, Washington, DC, USA, 1996. IEEE Computer Society.

[47] S. Forrest, A. S. Perelson, L. Allen, and R. Cherukuri. Self-nonself discrimination in a computer. In *In Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy*, pages 202–212. IEEE Computer Society Press, 1994.

[48] S. M. Garrett. How do we evaluate artificial immune systems. *Evolutionary Computation*, 13:145–178, 2005.

[49] J. Gausemeier, F.-J. Rammig, and W. Schäfer, editors. *Design Methodology for Intelligent Technical Systems - Develop Intelligent Technical Systems of the Future*. Springer-Verlag, Heidelberg, Germany, Jan. 2014.

[50] J. Gausemeier, F. J. Rammig, W. Schäfer, and W. Sextro, editors. *Dependability of Self-optimizing Mechatronic Systems*. Lecture Notes in Mechanical Engineering. Springer, Heidelberg New York Dordrecht London, 2014.

[51] J. Greensmith, U. Aickelin, and S. Cayzer. Introducing dendritic cells as a novel immune-inspired algorithm for anomaly detection. In *Proceedings of the 4th International Conference on Artificial Immune Systems*, ICARIS'05, pages 153–167, Berlin, Heidelberg, 2005. Springer-Verlag.

[52] J. Greensmith, U. Aickelin, and S. Cayzer. Detecting danger: The dendritic cell algorithm. *CoRR*, abs/1006.5008, 2010.

[53] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, New York, NY, USA, 1997.

[54] E. Hart and J. Timmis. Application areas of ais: The past, the present and the future. In *Proceedings of the 4th International Conference on Artificial Immune Systems*, ICARIS'05, pages 483–497, Berlin, Heidelberg, 2005. Springer-Verlag.

[55] E. Hart and J. Timmis. Application areas of ais: The past, the present and the future. *Appl. Soft Comput.*, 8(1):191–201, Jan. 2008.

[56] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6:151–180, 1998.

[57] S. A. Hofmeyr and S. A. Forrest. Architecture for an artificial immune system. *Evol. Comput.*, 8(4):443–473, Dec. 2000.

[58] G. Horányi, Z. Micskei, and I. Majzik. Scenario-based automated evaluation of test traces of autonomous systems. In M. Roy, editor, *Proceedings of ERCIM/EWICS Workshop on Dependable Embedded and Cyber-physical Systems (DECS) at SAFECOMP'13*, pages 181–192, Toulouse, France, 09/2013 2013.

[59] M. Kay. *Chapter 5, Suffix Trees and its Construction*. Course Readings, http://www.cbcb.umd.edu/confcour/Fall2012/suffixtrees.pdf, 2012.

[60] E. Keogh, J. Lin, and A. Fu. Hot sax: efficiently finding the most unusual time series subsequence. In *Data Mining, Fifth IEEE International Conference on*, pages 8 pp.–, Nov 2005.

[61] J. Kim, P. Bentley, U. Aickelin, J. Greensmith, G. Tedesco, and J. Twycross. Immune system approaches to intrusion detection – a review. *Natural Computing*, 6(4):413–466, 2007.

[62] J. Kim, J. Greensmith, J. Twycross, and U. Aickelin. Malicious code execution detection and response immune system inspired by the danger theory. In *Proceedings of the Adaptive and Resilient Computing Security Workshop (ARCS-05)*, Santa Fe, USA, 2005.

[63] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1st edition, 1997.

[64] S. Kumar and E. H. Spafford. An application of pattern matching in intrusion detection. Technical report, 1994.

[65] T. Lane and C. E. Brodley. Temporal sequence learning and data reduction for anomaly detection. *ACM Trans. Inf. Syst. Secur.*, 2(3):295–331, Aug. 1999.

[66] R. Langmann, editor. *Self-nonself discrimination revisited*, volume 12. 2000.

[67] N. Lay and I. Bate. Applying artificial immune systems to real-time embedded systems. In *Congress on Evolutionary Computation (CEC)*, 2007.

[68] N. Lay and I. Bate. Improving the reliability of real-time embedded systems using innate immune techniques. volume 1, pages 113–132. Springer-Verlag, 2008.

[69] S. Li. A Framework for Health Monitoring in the Real-Time Operating System ORCOS. Master thesis, Faculty of Computer Science, Electrical Engineering, and Mathematics, Paderborn University, 2014.

[70] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, Jan. 1973.

[71] F. Macias. The test and evaluation of unmanned and autonomous systems. *International Test and Evaluation Association, ITEA Journal*, 29:388–395, 2008.

[72] F. Maggi, M. Matteucci, and S. Zanero. Detecting intrusions through system call sequence and argument analysis. *Dependable and Secure Computing, IEEE Transactions on*, 7(4):381–395, Oct 2010.

[73] M. Markou and S. Singh. Novelty detection: A review - part 1: Statistical approaches. *Signal Processing*, 83:2003, 2003.

[74] M. Markou and S. Singh. Novelty detection: A review - part 2: Neural network based approaches. *Signal Processing*, 83:2499–2521, 2003.

[75] M. Masmano, I. Ripoll, P. Balbastre, and A. Crespo. A constant-time dynamic storage allocator for real-time systems. *Real-Time Systems*, 40(2):149–179, 2008.

[76] P. Matzinger. Tolerance, danger, and the extended family. *Annual Reviews on Immunology*, 12:991–1045, 1994.

[77] A. Mueen and E. Keogh. Online discovery and maintenance of time series motifs. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '10, pages 1089–1098, New York, NY, USA, 2010. ACM.

[78] S. Oberthür. *Towards an RTOS for self-optimizing mechatronic systems*. Phd thesis, Faculty of Computer Science, Electrical Engineering, and Mathematics, Paderborn University, Paderborn, Germany, 2010.

[79] A. Patcha and J.-M. Park. An overview of anomaly detection techniques: Existing solutions and latest technological trends. *Computer Networks*, 51(12):3448 – 3470, 2007.

[80] C. Priesterjahn. *Analyzing Self-healing Operations in Mechatronic Systems*. Dissertation, Fakultät für Elektrotechnik, Informatik und Mathematik, Universität Paderborn, Paderborn, Aug. 2013.

[81] QEMU Group. QEMU/Monitor on the Website of Wikibooks, 2014. http://en.wikibooks.org/wiki/QEMU/Monitor.

[82] U. Richter, M. Mnif, J. Branke, C. Müller-Schloer, and H. Schmeck. Towards a generic observer/controller architecture for organic computing. In C. Hochberger and R. Liskowsky, editors, *INFORMATIK 2006 ? Informatik für Menschen!*, volume P-93 of *LNI*, pages 112–119. Bonner Köllen Verlag, Oktober 2006.

[83] V. P. Roske, I. Kohlberg, and R. Wagner. Autonomous systems challenges to test and evaluation. Presented at the 28th Annual National Test and Evaluation Conference of National Defense Industrial Association, Mar. 2012.

[84] X. Song, M. Wu, C. Jermaine, and S. Ranka. Conditional anomaly detection. *IEEE Trans. on Knowl. and Data Eng.*, 19(5):631–645, May 2007.

[85] W. Stallings. *Operating Systems: Internals and Design Principles*. Prentice Hall Press, Upper Saddle River, NJ, USA, 6th edition, 2008.

[86] J. Stöcklein, W. Müller, D. Baldin, and T. Xie. Virtual test environment for self-optimizing systems. In *ASME/IEEE International Conference on Mechatronic and Embedded Systems and Applications (MESA2013)*. ASME, 4 - 7 Aug. 2013.

[87] P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining, (First Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[88] M. Thompson. Testing the intelligence of unmanned autonomous systems. volume 29. ITEA Journal of Test & Evaluation, Dec. 2008.

[89] J. Twycross and U. Aickelin. libtissue - implementing innate immunity. In *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*, pages 499–506, 2006.

[90] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[91] Unity Technologies. Unity3d game engine, 2014. http://unity3d.com.

[92] G. Vaz. A reconfigurable Real-time Monitoring Framework for a Real-time Operating System. Master thesis, Faculty of Computer Science, Electrical Engineering, and Mathematics, Paderborn University, 2013.

[93] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: alternative data models. In *Security and Privacy, 1999. Proceedings of the 1999 IEEE Symposium on*, pages 133–145, 1999.

[94] P. Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (Swat 1973)*, SWAT '73, pages 1–11, Washington, DC, USA, 1973. IEEE Computer Society.

[95] Y. Zhao and F.-J. Rammig. Online model checking for dependable real-time systems. In *16th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing-Shenzhen, China*, pages 154–161. IEEE Computer Society, IEEE Computer Society, 11 - 13 Apr. 2012.