UNIVERSITY OF PADERBORN

Online Model Checking Mechanism and Its Applications

by

Yuhong Zhao

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

in the
Faculty of Electrical Engineering, Computer Science, and Mathematics
University of Paderborn

January 2016

"Make things as simple as possible, but not simpler."

(*attributed to*) Albert Einstein

Abstract

Modern embedded systems are a type of special-purpose computer systems. They are widely used in industry and are becoming increasingly complicated due to the advances in electronic techniques. Design errors in software account for a large percentage of the computer system failures. In this thesis, we concern ourselves with online checking the correctness of the control software applied to such kind of embedded systems that are identified as safety-critical, whose failure or malfunction may cause severe damages.

The existing validation and verification techniques can not completely ensure that the embedded software does behave as desired after it is released or deployed. Against this background, we present an online model checking mechanism aimed to ensure the correctness of the actual execution trace, instead of the universal correctness, of the embedded software system. Notice that we don't mean to propose a faster model checking algorithm. The basic idea is to check during system execution a sequence of bounded models that cover the actual execution trace of the software system under investigation. Errors detected in the bounded models may indicate potential errors in the source code of the target system. The bounded models are derived from the behavioral model of the target system using the actual state information monitored periodically during system execution. The online model checking problem is reduced to online reachability analysis, which tries to look ahead finitely many steps on the model level. The properties to be checked are specified in linear temporal logic. Because the checking process is done on the model level, both safety and liveness properties can be handled during runtime.

By doing model checking online, we are able to reach those states that locate arbitrarily deep in the state space and to predict potential errors even if the checking process falls behind the execution of the target system. The state space explosion problem can thus be avoided to some degree because the models to be checked are always bounded ones. However, doing model checking online has to suffer from the limited execution time allocated to each checking cycle. To deal with this problem, we speed up online reachability analysis by reducing workload and adopting the symbolic state-based search algorithm as well as using parallel computing. We present a general framework for integration of online model checking with a real-time operating system, which can be implemented on different hardware architectures from single-core, or multi-core to multiprocessor. The RA component of the TCAS software is taken as case study to demonstrate the applicability of our online model checking method. In addition, we extend the application of the online model checking mechanism to hybrid systems.

Zusammenfassung

Moderne eingebettete Systeme sind spezielle Computersysteme. Sie sind in der Industrie weit verbreitet und als Ergebnis der Fortschritte der Halbleitertechnologie immer komplexer geworden. Designfehler in Software machen einen großen Prozentsatz der Fehler in Computersystemen aus. In dieser Arbeit befassen wir uns mit der Online Überprüfung der Korrektheit von Kontrollsoftware in eingebetteten Systemen, die als sicherheitskritische identifiziert sind.

Die vorhandenen Validierungs- und Verifikationstechniken können nicht vollständig sicherstellen, dass sich die eingebettete Software wirklich wie gewünscht verhält, nachdem sie freigegeben oder eingesetzt wurde. Vor diesem Hintergrund stellen wir einen Online Model Checking Mechanismus vor, um die Korrektheit eines aktuellen Ausführungspfades, anstatt die gesamte Korrektheit der eingebetteten Software, sicherzustellen. Es ist dabei nicht das Ziel, einen schnelleren Model Checking Algorithmus vorzulegen. Die Grundidee des Ansatzes ist es, eine Folge von partiellen Modellen, die den aktuellen Ausführungspfad der zu überprüfenden Software überdecken, während der Systemausführung zu überprüfen. Die Fehler, die in den partiellen Modellen erkannt werden, können mögliche Fehler im Quellcode des zu überprüfenden Systems anzeigen. Die partiellen Modelle entstehen aus dem Verhaltensmodell des zu überprüfenden Systems mittels der aktuellen Zustandsinformation, die während der Laufzeit periodisch aufgenommen wird. Das Online Model Checking Problem reduziert sich zu Online Erreichbarkeitsanalyse, wobei in jedem Überprüfungszyklus nur endlich viele Schritte auf der Modellebene verfolgt werden. Die zu überprüfenden Eigenschaften sind Formeln in Linearer Temporaler Logik. Sowohl Sicherheits- wie auch Lebendigkeitsüberprüfungen lassen sich dabei auf Erreichbarkeitsanalyse während der Laufzeit zurückführen.

Mittels Online Model Checking sind wir in der Lage, die Zustände, die sich beliebig tief in dem Zustandsraum befinden, zu erreichen. Dazu können wir auch potenzielle Fehler vorhersagen, selbst wenn der Checking Prozess hinter der Ausführung des zu überprüfenden Systems zurückfällt. Das Problem der Zustandsraumexplosion kann zu einem gewissen Grad vermieden werden, da Online Model Checking eine vereinfachte Form von Bounded Model Checking mit geleitenden initialen Zuständen ist, welches zur Laufzeit angewandt wird. Andererseits leidet Online Model Checking unter der beschränkten Ausführungsfrist, die für jeden Überprüfungszyklus festgelegt wird. Um dieses Problem zu lösen, beschleunigen wir die Online-Erreichbarkeitsanalyse durch gezielte Verringerung der Arbeitsbelastung und die Verwendung eines symbolischen zustandsbasierten Suchalgorithmus sowie mittels Parallel Computing. Wir präsentieren einen allgemeinen Rahmen für die Integration von Online Model Checking in ein Echtzeitbetriebssystem. Dieser Ansatz kann auf unterschiedlichen Hardware-Architekturen, von Single-Core- oder Multi-Core- bis hin zu Multiprozessor-Architekturen, implementiert werden. Die RA-Komponente der TCAS Software dient als Fallstudie, um die Anwendbarkeit unserer Online Model Checking Methode zu demonstrieren. Darüber hinaus erweitern wir die Anwendung des Online Model Checking auf Hybridsysteme.

Acknowledgements

Many people have provided help and encouragement over these years, without which this thesis would not have been possible.

I would like to express my special appreciation and thanks to Prof. Franz Rammig for accepting me in his research group at the time when I was getting into trouble in China. Words can not express how grateful I am to him! His confidence and sustainable support encourage me to keep going and continue my research on this topic.

I am grateful to Prof. Uwe Glässer, Prof. Hans Kleine Büning, Prof. Heike Wehrheim, and Jun. Prof. Christian Plessl for serving on my examination committee.

I am thankful to Prof. Gabor Karsai for reviewing this thesis.

My thanks go to Krishna Sudhakar, Mona Qanadilo, Sufyan Samara, Karsten Scheibler, and Kathrin Flaßkamp. Some ideas would not be implemented without their help.

I would also like to thank Peter Schrammel for helping me understand the CBMC tool and Ian Mitchell for helping me learn the level set methods.

A particular thank goes to my husband for bringing me pleasure. With him my life becomes simple and is full of happiness so that I can overcome any interference occurred in writing this thesis.

I am deeply grateful to my parents. Without them I would not be where I am today.

I would like to take this opportunity to thank Mr. Li Hongzhi for teaching me the principle of "Truthfulness, Compassion, and Tolerance", whereby I am able to maintain a peaceful mind in everyday life, be it at work or at home.

Contents

Chapter 1

Introduction

Nowadays *embedded systems* are playing an increasingly important role in our daily life. Embedded systems are computer systems integrated into technical products, such as flight control, automotive drive-by-wire, nuclear reactor management, and others. The advances in electronic techniques enable the hardware of embedded systems to run highly sophisticated software. Therefore, more functionality can be implemented in software. E.g., modern cars usually have 20 to 70 electronic control units (ECUs) with millions of lines of code [1]. More specifically, the engine control unit, the most powerful ECU on most cars, executes concurrently up to 100 (software) tasks [2].

Autonomous and Autonomic Systems

By observing the evolution of cars in the past several decades, we are able to envision that the development trends for embedded systems are moving from automated towards *autonomous* and *autonomic* systems. E.g., driverless cars are capable of sensing the environment and navigating without human intervention. In 2012, Google cofounder Sergey Brin said that Google will have autonomous cars available for the general public within five years [3]. Amazon is now exploring the use of drones, a kind of unmanned aerial vehicle (UAV), for its package delivery service [4].

It follows that new software technology tends to add *autonomy* to modern embedded systems so that they are able to operate on their own with little or even no directions from humans [5]. Autonomous and autonomic are the two aspects of autonomy: autonomous indicates self-directed to make the system fulfill some goal(s) independently, whereas autonomic implies self-managing to keep the system robust against adversarial impacts, no matter what happens in the environment.

In academia, more attention is paid to autonomic computing [6], which aims to make embedded systems capable of managing themselves in response to changes in the system objectives or in the environment by means of self-configuration, self-optimization, self-healing and self-protection. E.g., the RoSES project [7] proposes a general approach to building robust distributed embedded systems capable of configuring themselves by adding or removing components in the field rather than in the factory. The project of Collaborative Research Centre (CRC) 614 "Self-Optimizing Concepts and Structures in Mechanical Engineering" [8] presents a design methodology for tomorrow's self-optimizing electro-mechanical systems whose behaviors are characterized by the communication and cooperation between the components with inherent "intelligence". Here the self-optimization is implemented by means of changing the parameters or the structure of the system components. In addition, an organic programming approach [9] is presented for cyber-physical systems capable of self-adapting to changing environments.

Safety Problems

Many embedded systems are safety-critical and long-lived systems. For safety-critical systems, failures may cause high costs and even endanger human lives. In general, a (distributed) computer system may fail due to external and/or internal reasons [2]. External reasons are related to the system specification itself or to the operational environment, e.g., mechanical stress, wrong input, temperature, and so on. The main internal reasons for failure may be: (i) random physical faults in hardware; (ii) design faults in hardware and/or software; or (iii) communication failures in a distributed environment.

According to [2], "Field data on the observed reliability of many large computer systems indicates that a significant and increasing number of computer system failures are caused by design errors in the software and not by physical faults of the hardware."

Software errors are caused usually by the unmanaged complexity of the system design. The increasing complexity of the embedded software makes subtle errors extremely difficult to figure out or to reproduce in a laboratory environment. Although safety-critical systems are usually designed to be fault-tolerant, experience shows that software errors are still unavoidable.

We have to mention this widely known accident. On June 4, 1996 the Ariane 5 launcher went into self-destruction mode 37 seconds after liftoff. The failure was caused by a software error in the inertial reference system: a 64 bit floating point number (representing the horizontal velocity) was converted to a 16 bit signed integer. Consequently, the conversion failed and the guidance and altitude information was lost. Indeed the program was the same as the one that had worked perfectly in Ariane 4, while the continuous dynamical systems around the software had changed. In the new physical environment, the trusted code unfortunately led to a catastrophe [10].

In 2005 a Boeing 777-120 aircraft experienced an in-flight upset event due to a software design error [11]. According to the investigation conducted by the Australian Transport Safety Bureau, the problem stemmed from an error in the ADIRU[1] software. The error had existed in previous releases of the ADIRU software, but had been masked by other code. The error was eventually exposed by a series of events that was unlikely to have been revealed in the testing and certification process for the unit [12]. Due to the software error, the fault-tolerant software used the erroneous data to make wrong decision.

David Cummings described also a case study they encountered in checking the flight software for NASA's Mars Pathfinder spacecraft [13]. A simple test which should produce an even result (2, 4, 6, and so on) was inserted into the software. They observed just once that the check had failed. They were "never able to reproduce the failure, despite repeated attempts over many thousands if not millions of iterations."

Another case study is about Toyota's unintended acceleration problem [14]. Nowadays cars' throttles are mostly electronic instead of mechanical. Between the sensor under the gas pedal and the actuator in the fuel injector many things are likely to go wrong in a "drive-by-wire" system. It's been reported since 2009 that Toyota Corollas can accelerate unexpectedly at low speeds. A careful examination of the car's software (i.e., firmware) indicated that it could have failed in the way described in the case, not necessarily that it did fail [15]. Toyota's engine-control code contains more than 11,000 global variables. The program structure is very complex. Various studies over the years determined that functions with a *cyclomatic complexity*[2] of greater than 10 have a higher risk of defects [16]. Many functions in Toyota's code have a cyclomatic complexity of higher than 50. In particular, the cyclomatic complexity of throttle-angle sensor function is more than 100. It is really difficult to check and ensure the correctness of such a complicated software.

Needless to say, it is quite important to ensure the correctness of embedded software. Unfortunately, no existing verification and validation techniques can completely ensure that a software system does behave as desired after it is released or deployed.

Existing Solutions: Testing, Model Checking, and Online Monitoring

For industrial designs *testing* is the mainstream solution to the safety problem of modern embedded software. Software testing [17] tries to find defects by executing a program

[1]An acronym for Air Data Inertial Reference Unit.
[2]An integer-based metric used to measure the complexity of a program by counting the number of linearly independent paths through the program.

to see whether the required results are met or not. There is no way to completely test a program of a moderate complexity. For untested inputs, undiscovered errors in deep corners may show up during system execution. Even if an error is found by testing, it is usually difficult to figure out the reason(s) for the error.

Different *model checking* techniques [18] play a supporting role in ensuring the safety of large complex systems. Model checking needs to explore exhaustively the state space of the behavioral model of the software system under investigation within reasonable time and memory consumption. The complexity of autonomous and autonomic systems exacerbates the state space explosion problem. Even the reachability problem can not be solved completely when the state space is too large.

To challenge the state space explosion problem, many effective methods have been proposed in the literature: *partial order reduction* [19], *compositional reasoning* [20], *abstraction interpretation* [21], *bounded model checking* [22], to name just a few. Partial order reduction exploits the commutativity of the executed transitions in asynchronous systems, which results in the same states when executed in different orders, to reduce the state space that needs to be searched. Compositional verification is a divide-and-conquer approach to mitigating the state space explosion in concurrent systems. Assumptions on the environment are needed to guarantee the correctness of the individual components. It is usually a non-trivial task to find the right assumptions and to check the refinement of the environment against the assumptions. Abstraction is usually used in the Counterexample Guided Abstraction Refinement (CEGAR) paradigm [23], where the abstraction of the behavioral model is refined iteratively until either a definite result is obtained or the refined model becomes intractable. Bounded Model Checking (BMC) tries to search for an error path of length up to some finite bound from initial states. Theoretically, there does exist a *completeness threshold* [24] for the bound, but it is usually too large to perform BMC up to this threshold in practice.

Without doubt these improvements do make model checking applicable to more complex systems, but at the cost of making the checking process more complicated and thus error-prone, because the correctness of the checking program itself is difficult to be verified exhaustively.

Note that the above mentioned checking techniques and the like are traditionally applied during the software development phase before the software system is deployed in the field. In this sense, they belong to the *offline* verification category.

Of course, there exist also other offline checking techniques, such as theorem proving, static analysis, simulation, and the like. All these offline checking methods fail to ensure definitely the correctness of large complex systems during the system development phase.

Therefore, different *online monitoring* techniques (see Section 3) come into play. The concept of online monitoring dates back to the assembly language era. Online monitoring has been used in program debugging, profiling, optimization, and so on. Nowadays online monitoring is applied to checking for the correctness of the actual execution trace of the (software) system under observation. In this sense, it is also called *runtime verification*.

The basic idea of online monitoring is to observe the state information while the target system is running, and then analyze based on the collected data whether the target system behaves normally with respect to the given properties. In practice, the analysis can be carried out on the spot or at some time later. The granularity of observation may have a large impact on online monitoring. If it is too coarse, important information may be missed; if it is too fine, the monitoring overhead will be too high. On the other hand, the properties to be checked are usually derived from the system requirements, whereas the execution trace to be monitored is at the code (i.e., implementation) level. The *semantic gap* between the properties and the execution trace makes it usually difficult to connect correctly the low level events (i.e., state information) with the corresponding high level elements in the properties.

Our Solution: Online Model Checking

Given the source code and the (behavioral) model of the software program as well as the properties to be checked, obviously, the semantic gap between the properties and the behavioral model is more narrow than the semantic gap between the properties and the source code of the target program. The behavioral model describes the system behaviors at a higher level of abstraction. It may be generated based on the system requirements or extracted from the source code of the target system. Anyway, it is relatively easier to establish the semantic relationship between the source code and the behavioral model of the target system, i.e., the mapping function from the low-level concrete states to the high-level abstract states, as well as the semantic relationship between the behavioral model and the properties to be checked. In other words, the behavioral model can bridge the semantic gap between the source code and the properties to be checked.

To a fairly large degree, the correctness of the behavioral model can reflect the correctness of the (source code) implementation of the target system. Because many implementation details are abstracted away, the behavioral model is usually much simpler than the source code. However, for large complex systems, such as autonomous and autonomic systems, the behavioral model is still too complex to be explored exhaustively by offline checking techniques.

Against this background, we present the concept of online model checking, which is the contribution of this thesis. Online model checking can be seen as an extension of online monitoring. Like online monitoring, online model checking also observes the state information of the target system at runtime and then checks the correctness of the current execution trace against the given properties. Unlike online monitoring, online model checking does not try to figure out potential errors from the collected data. Instead, it tries to identify a partial model of the target program based on the observed state information and then to search for errors in the obtained partial model. Errors found in the partial model may indicate potential errors in the current execution trace or even predict errors that may happen in the near future. The counterexample produced by means of online model checking can be used to discover the root cause of the errors.

Considering that model checking is usually a time consuming process, it seems to be an "impossible mission" to do model checking online while the target system is running. This thesis tries to give a possible solution following the principle of making things "as simple as possible, but not simpler"[3]. On the one hand, compared with (offline) model checking, online model checking is obviously *simple*; on the other hand, compared with online monitoring, online model checking is *not* that much *simpler*.

It is worth mentioning that online model checking is a complementary technique to the existing solutions. It is originally proposed to be used during runtime after the target program is deployed so as to provide an additional defense mechanism against potential design errors in the program. Of course, it can also be used as an aid in software testing to improve the test coverage, but this topic is outside the scope of this thesis.

Thesis Organization

The remainder of this thesis is organized as follows: Chapter 2 explains the basic concepts and techniques used in the thesis to make the thesis self-contained; Chapter 3 details the online monitoring techniques presented in the literature and discusses the differences and similarities between online monitoring and online model checking; Chapter 4 presents our online model checking mechanism as well as different speed-up techniques to improve the performance of online model checking; Chapter 5 proposes a lightweight method to decide the monitoring points in the target program; Chapter 6 integrates online model checking with an RTOS as a verification service and analyzes the monitoring overhead as well as the communication overhead; Chapter 7 provides a case study to demonstrate the applicability of online model checking; Chapter 8 extends the application of online model checking to hybrid systems; finally, we draw conclusions and point out possible future research directions in Chapter 9.

[3]attributed to Albert Einstein.

Chapter 2

Preliminaries

To make the thesis self-explanatory, the following sections provide the definitions of the terms used throughout this thesis.

2.1 Fault, Error and Failure

There are no unique and commonly accepted precise definitions of the concepts fault, error and failure in the literature related to dependable computing. Here we adopt the definitions of fault, error and failure proposed by Avizienis et al. in [25].

A *failure* indicates that some externally observable state of a system deviates from the intended one, provided that there is output from the system; otherwise, there is no failure, even if something does go wrong inside the system. That is, a failure refers to some misbehavior of the system that can be observed by the user, be it a human user or another computer system.

In contrast, an *error* indicates that some internal state of the system deviates from the desired one, i.e., something goes wrong inside the system. Error(s) may or may *not* result in failure(s).

A *fault* is "the adjudged or hypothesized cause of an error", which can be internal or external relative to the system under investigation.

Let's take an example presented in [26] to further clarify the terms fault, error, and failure at the software level. Given a system with the fault of missing the **free** statement in its program, whenever the piece of code that should free memory is executed, the program enters an error state: memory is allocated but never released. As long as the consumed memory keeps below a certain threshold, there is no failure observable from outside. However, once the memory limits of the system are approached, a failure is observed.

2.2 Behavioral Model

The behaviors of a computer system can be modeled at various levels, e.g., at four levels from low to high: the *physical* level, the *digital logic* level, the *information* level and the *external* level [27]. In this thesis our interest is in the behaviors of a computer system at the information level, i.e., software system. We'd like to detect errors in a software system by doing model checking online while the software system is running. For this purpose, we need to specify formally a behavioral model in terms of states and transitions so as to reflect the computations of the software system at some abstraction level and/or from some perspective.

While a software system is running, the values of the variables of the software system will be updated over time. A *state* captures the values of the system variables at a particular instant of time. Given a state, the values of the system variables can be changed by executing an action, which results in an evolution of the system from the current state to a next state. Such a pair of states (before and after the action is executed) indicates a *transition* of the software system. A *run* of the system can thus be defined as a sequence of (possibly infinite) states connected by transitions.

Let $V = \{v_1 : D_1, v_2 : D_2, \cdots, v_n : D_n\}$ be the set of the system variables $v_1, v_2, \cdots, v_n$ ranging over the finite domains $D_1, D_2, \cdots, D_n$ respectively. A state is just a *valuation* $s : V \to \{D_1, D_2, \cdots, D_n\}$ for the variables in V. Given a valuation (i.e., a state) $\langle v_1 = d_1, v_2 = d_2, \cdots, v_n = d_n \rangle$ with $v_i \in V$ and $d_i \in D_i$ for $i = 1, 2, \cdots, n$. We can use the formula $(v_1 = d_1) \wedge (v_2 = d_2) \wedge \cdots \wedge (v_n = d_n)$ to represent it, where each proposition $v_i = d_i$ is regarded as an atomic, basic element. Generally, atomic propositions have the form $v_i = d_i$. An atomic proposition $v_i = d_i$ is *true* in a state s if $s(v_i) = d_i$. It is easy to see that each state indicates a set of atomic propositions *true* in this state. On the other hand, there may be more than one state in which the atomic proposition $v_i = d_i$ holds. Hence, a formula can be interpreted as a set of *all* such states that make it *true*.

Similarly, a transition (s, s') is a valuation of variables in the current state s and in the next state s'. To distinguish the variables in the current state from the ones in the next state, we rename the variables in the next state as $V' = \{v'_1 : D_1, v'_2 : D_2, \cdots, v'_n : D_n\}$. Thus, for each variable v_i in V, there is a corresponding (next state) variable v'_i in V'. Now we are able to represent a set of transitions using a logic formula, too. This formula is called *transition relation*, denoted as $R(V, V')$.

Given a software program P, the program variables and their value domains are available. Let AP be the set of atomic propositions. Formally, the behaviors (or computations) of the software program over AP can be modeled as a kind of state transition graph called *Kripke* structure, denoted as $M = (V_M, D_M, R_M, I_M, L_M)$, where

- $V_M = \{v_1 : D_1, v_2 : D_2, \cdots, v_n : D_n\}$ declares the set of the system variables and their corresponding value domains,
- $D_M = D_1 \times D_2 \times \cdots \times D_n$ defines the state space of M,
- $R_M \subseteq D_M \times D_M$ is the transition relation of M,
- $I_M \subseteq D_M$ is the initial condition of M, and
- $L_M : D_M \rightarrow 2^{AP}$ is a labeling function that associates each state with the set of atomic propositions (in AP) *true* in that state.

In the state space of M, there is a transition between two states s and s', if $R_M(s, s')$ holds. For the sake of convenience, we suppose the transition relation R_M to be *total.* That is, for every state $s \in D_M$ there exists a state $s' \in D_M$ such that $R_M(s, s')$ holds. In fact, we can always make R_M total by adding an auxiliary transition to each state s without successors so that $R_M(s, s)$ holds.

A *path* (or *run*) ρ of M from a state s is an infinite sequence of states $\rho = s_0, s_1, s_2, \cdots$ such that $s_0 = s$ and $R_M(s_t, s_{t+1})$ holds for all $t \geq 0$. Let S_0 be the set of initial states of M, i.e., $S_0 = \{s \in D_M \mid s \models I_M\}$. A state s is *reachable*, if there exists a path ρ with some prefix $s_0, s_1, \cdots, s_t$ in M such that $s_0 \in S_0$ and $s_t = s$. Notice that not all of the states in D_M are reachable from the given initial states. The set of reachable states of M, denoted as S, are those states in D_M reachable from the set S_0 of initial states. For large complex systems, especially parallel systems, it is usually difficult to identify all the reachable states of the system model. In general, it is undecidable whether a state of the system model is reachable or not [28].

Let's take a simple transition system described in [18] as an example to explain the above notions. The *Kripke* structure M of the simple system is defined as follows:

- $V_M = \{x : D, y : D\}$ where $D = \{0, 1\}$,
- $D_M = D \times D = \{(1, 1), (0, 1), (1, 0), (0, 0)\}$,
- $R_M(x, y, x', y') \equiv x' = (x + y) \ mod \ 2 \wedge y' = y$,
- $I_M \equiv x = 1 \wedge y = 1$, and

- $L_M((1,1)) = \{x = 1, y = 1\}$, $L_M((0,1)) = \{x = 0, y = 1\}$, $L_M((1,0)) = \{x = 1, y = 0\}$, and $L_M((0,0)) = \{x = 0, y = 0\}$.

Fig. 2.1 illustrates a graphical representation of the *Kripke* structure of this example. The state space of the system consists of four states $(1,1)$, $(0,1)$, $(1,0)$, and $(0,0)$. The initial state $(1,1)$ is pointed to by an incoming edge without a source. It is easy to see that the states $(1,0)$ and $(0,0)$ are not reachable from the initial state $(1,1)$. The path $\rho = (1,1),(0,1),(1,1),\cdots$ is the only path that starts from an initial state. This path is the only valid behavior (or computation) of the transition system.

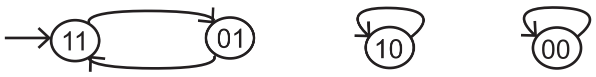

FIGURE 2.1: A *Kripke* structure example

2.3 Property Specification

Property specification defines formally a set of implementation-independent *constraints* that a software system under investigation needs to satisfy. The properties are usually specified in some logical formalism, e.g., temporal logic, automata, or regular expression, etc., which describes how the behaviors of the software system should evolve over time. It is usually difficult to decide whether or not a given specification is *complete*, i.e., it covers all the properties that the software application should satisfy. It is also hard to prove that what we write does capture exactly what we mean. In this thesis, we simply suppose that the properties are correctly specified. Therefore, a model or an implementation of the software system is proved to be *correct*, it means implicitly that it is correct with respect to the given property specification.

2.3.1 Linear Temporal Logic

In this thesis, our concern is the correctness of the individual executions of the software program under investigation. Therefore, linear temporal logic (LTL) [29] is adopted to specify the properties that the software system is required to satisfy. LTL can specify the ordering of states in time without defining time explicitly.

Given a behavioral model M over AP of the software program to be checked, an LTL property over AP has the form $\mathbf{A}f$, where $\mathbf{A}$ is a path quantifier meaning "for all paths", and f is a path formula specifying a (characteristic) predicate on the paths of the model

M. For technical convenience, we consider LTL formulas in *positive normal form*, i.e., negations (if any) are applied only to atomic propositions. An LTL formula f in positive normal form is inductively defined as follows:

- *true* and *false* are LTL formulas;
- if $p \in AP$, then p and $\neg p$ are LTL formulas;
- if x and y are LTL formulas, then $x \vee y$, $x \wedge y$, $\mathbf{X}y$, $\mathbf{F}y$, $\mathbf{G}y$, $x\mathbf{U}y$, and $x\mathbf{R}y$ are LTL formulas, where **X** (neXt), **F** (Future), **G** (Global), **U** (Until) and **R** (Release) are temporal operators.

Let $\rho = s_0, s_1, \cdots, s_i, s_{i+1}, \cdots$ be a path in the model M. We define the suffix of ρ starting from s_i as $\rho^i = s_i, s_{i+1}, \cdots$. The semantics of an LTL formula with respect to the path ρ in M is formally defined below [18]:

1. $M, \rho \models p$ iff $p \in L_M(s_0)$.
2. $M, \rho \models \neg p$ iff $p \notin L_M(s_0)$.
3. $M, \rho \models x \vee y$ iff $M, \rho \models x$ or $M, \rho \models y$.
4. $M, \rho \models x \wedge y$ iff $M, \rho \models x$ and $M, \rho \models y$.
5. $M, \rho \models \mathbf{X}y$ iff $M, \rho^1 \models y$.
6. $M, \rho \models \mathbf{F}y$ iff $M, \rho^i \models y$ for some $i \geq 0$.
7. $M, \rho \models \mathbf{G}y$ iff $M, \rho^i \models y$ for all $i \geq 0$.
8. $M, \rho \models x\mathbf{U}y$ iff $M, \rho^j \models y$ for some $j \geq 0$ and $M, \rho^i \models x$ for all $0 \leq i < j$.
9. $M, \rho \models x\mathbf{R}y$ iff for all $j \geq 0$, if for every $i < j$, $M, \rho^i \not\models x$ then $M, \rho^j \models y$.

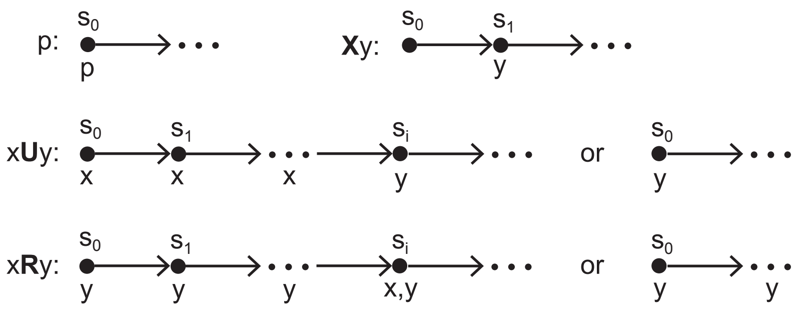

FIGURE 2.2: The visualized explanation of some LTL formulas

Intuitively, an LTL property defines the characteristics of those valid paths by imposing certain constraint(s) on some placeholder(s) in the paths as illustrated in Fig. 2.2. For example, the formula p requires that p holds in the first state of the path, $x\mathbf{U}y$ specifies that x holds along the path until in some states where y holds, while $x\mathbf{R}y$ specifies that y holds along the path up to and including the first state where x holds. In the latter two cases, x is not required to hold eventually.

We usually go to check the negation of an LTL property, which describes the characteristics of those error paths. In a model with finitely many states, an *infinite* path is represented as a *finite* sequence of states of the form $(s_0, s_1, \cdots, s_{i-1})(s_i, \cdots, s_n)$ with $s_i = s_n$ for $i \geq 0$, where $(s_0, s_1, \cdots, s_{i-1})$ is a *finite* prefix in case of $i > 0$ (or *empty* otherwise), and $(s_i, \cdots, s_n)$ an ending loop (i.e., *infinite* suffix) of the path. Therefore, an error path (counterexample) can be identified either by its finite prefix (finite witness) or by its ending loop (infinite witness) that satisfies some specific constraint(s) derived from the LTL formula.

There are two basic types of properties in LTL: *safety* and *liveness*. A general LTL property can be expressed as conjunction of a safety property and a liveness property [30]. Informally, a safety property means that something "*bad*" does never happen during program execution, while a liveness property means that something "*good*" does eventually happen [31].

Let $\Sigma = 2^{AP}$. Then, Σ^* represents the set of *finite* sequences of states, and Σ^ω the set of *infinite* sequence of states. For any two *finite* paths $\alpha, \beta \in \Sigma^*$, β^ω denotes the sequence of states obtained by *infinitely* repetition of β, and $\alpha \cdot \beta^\omega$ the *infinite* sequence of states obtained by concatenation of α and β^ω. Formally, an LTL formula f over AP is a safety property, if and only if $(\forall \alpha \in \Sigma^\omega.\ \alpha \models f) \iff (\forall i \geq 0.\ (\exists \beta \in \Sigma^\omega.\ \alpha[0..i] \cdot \beta \models f))$; f is a liveness property, if and only if $(\forall \alpha \in \Sigma^*.\ (\exists \beta \in \Sigma^\omega.\ \alpha \cdot \beta \models f))$ [30].

As for a safety property, there exists no specific constraint(s) to the *infinite* suffix β of the path satisfying it. That is, a counterexample of a safety property is a path with a *finite* prefix α whose last state contradicts the property. As for a liveness property, some desired state(s) must happen *infinitely* often along the path satisfying it. That is, a counterexample of a liveness property is a path ending with an *infinite* suffix β that contains no desired state(s).

For example, the safety property $\mathbf{G}p$, as shown in Fig. 2.3 (a), requires that something good (p) always holds (or something bad ($\neg p$) never holds); the liveness property $\mathbf{FG}p$, as shown in Fig. 2.3 (b), means no matter what happens along a finite prefix, eventually something desired (p) will happen infinitely often.

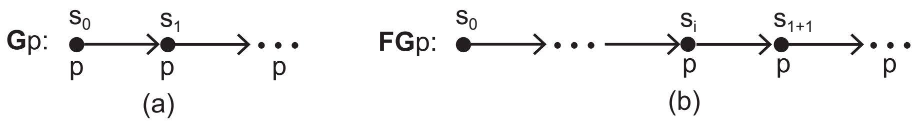

FIGURE 2.3: (a) Safety property $\mathbf{G}p$ and (b) Liveness property $\mathbf{FG}p$

2.3.2 *Büchi* Automaton

Given an LTL property f, we usually first transform its negation $\neg f$ into an equivalent *Büchi* automaton denoted as $B_{\neg f}$, and then go to check whether $B_{\neg f}$ is satisfiable with respect to the given model M, or not. The size of $B_{\neg f}$ is, in the worst case, exponential in the number of the subformulas of f. There are many algorithms in the literature [32–37] that generate optimized *Büchi* automata from LTL formulas. In practice, the commonly used requirements are not very sophisticated [38]. Indeed, the LTL formulas and their *Büchi* automata are usually not that much complicated.

Of course, we can also directly specify properties using *Büchi* automata. In fact, *Büchi* automata are more expressive than most temporal logic specification languages [39]. However, complementing directly a *nondeterministic Büchi* automaton involves an exponential blow-up [40]. In this sense, we prefer LTL to *Büchi* automata.

Informally, a *Büchi* automaton is an extension to a *finite* automaton in terms of acceptance condition. A finite automaton is a type of state transition graph with transitions labeled with *symbol*(s) and some states marked as accepting states. An accepting *run* of a finite automaton is a *finite* path from a start state to some accepting state. Instead, an accepting *run* of a *Büchi* automaton is an *infinite* path from a start state to an ending loop containing some accepting state(s) in it. In effect, the accepting state appears along the path *infinitely* often.

Let $\Sigma = 2^{AP}$ be the set of atomic propositions (or symbols in the context of automaton). Formally, a *Büchi* automaton B over Σ is defined as $B = (Q, \Delta, Q_0, F)$, where Q is a finite set of states; $\Delta : Q \times \Sigma \rightarrow Q$ is a transition function; $Q_0 \subseteq Q$ is a set of initial states; and $F \subseteq Q$ is a set of accepting states.

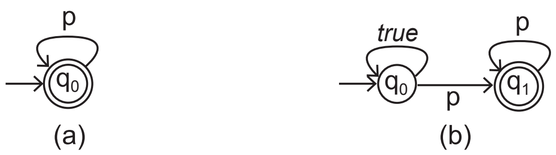

FIGURE 2.4: (a) $B_{\mathbf{G}p}$ and (b) $B_{\mathbf{FG}p}$

For example, Fig. 2.4 shows two *Büchi* automata $B_{\mathbf{G}p}$ and $B_{\mathbf{FG}p}$ generated from the LTL formulas $\mathbf{G}p$ and $\mathbf{FG}p$ respectively. In *Büchi* automata, the start states are denoted by an edge without a source and the accepting states by concentric circles. The transitions between states are labeled by predicates (i.e., symbols).

Let M be a model and B_f a *Büchi* automaton generated from an LTL formula f (with respect to M). An infinite *path* $\rho = s_0, s_1, \cdots, s_i, s_{i+1}, \cdots$ of M is accepted by B_f, i.e., $M, \rho \models f$, if and only if there exists a *run* $\gamma = q_0, q_1, \cdots, q_i, q_{i+1}, \cdots$ in B_f such that

(i) q_0 is a start state of B_f;

(ii) B_f moves from the state q_i to the next state q_{i+1}, if the state s_i in M satisfies the predicate on the transition (q_i, q_{i+1}); and

(iii) some accepting state of B_f appears in γ infinitely often.

Here we use "path" in the model M and "run" in the *Büchi* automaton B_f to distinguish the two sequences of states in different contexts. In addition, we also say that the path ρ *match*es the run γ in case that the first two conditions hold. Since the *Büchi* automaton B_f is usually nondeterministic, it is possible that the same path of M may match more than one run in B_f.

Note that if at some state q_i of B_f, the state s_i in M meets no predicate on any transition emanating from q_i, then we say that there is an *undefined* transition emanating from q_i.

It is easy to see that the *Büchi* automaton $B_{\mathbf{G}p}$ (resp. $B_{\mathbf{FG}p}$) in Fig. 2.4 accepts exactly the paths that satisfy the LTL formula $\mathbf{G}p$ (resp. $\mathbf{FG}p$). In addition, at the state q_0 in $B_{\mathbf{G}p}$ as well as at the state q_1 in $B_{\mathbf{FG}p}$ there exists (implicitly) an undefined transition with the predicate $\neg p$ on it.

Recall that a (nontrivial) LTL property is either safety or liveness or a conjunction of a safety property and a liveness property [30]. It is useful to distinguish between safety and liveness so that the model checking algorithms can deal with them in different efficient ways. We can use *Büchi* automata to determine whether a property is safety or not.

Let f be an LTL property and B_f a *Büchi* automaton[1] generated from f. Recall that a safety property claims that something "bad" does *never* happen, this is equivalent to saying that any *infinite* run in B_f is accepting, i.e., no "bad thing" occurs. There is no additional constraint(s) on the accepting states of the *Büchi* automaton. If a "bad" thing does happen, there is no way to remedy it; otherwise, it follows that something "good" would eventually happen. This means that a *finite* prefix is sufficient to contradict f in case that f is a safety property. This finite run in B_f must eventually end with some *undefined* transition. We call a finite *run* in B_f *bad* prefix (or "bad thing"), if there is no

[1]In B_f those states from which no accepting state is reachable are redundant, and thus deleted.

way to extend this finite run to an infinite run in B_f. In other words, there is no way to remedy this "bad thing". Consequently, f is a safety property, if B_f has bad prefix(es) but no constraint(s) on the accepting states, i.e., any infinite run in B_f is accepting.

Formally, a finite prefix $\alpha \in \Sigma^*$ in B_f is a bad prefix, if and only if $\forall \beta \in \Sigma^\omega.\ \alpha \cdot \beta \nvDash f$. That is, there is no way to extend α to an infinite run in B_f.

From $B_{\mathbf{G}p}$ it is easy to know that $\mathbf{G}p$ is a safety property. Let's introduce a special state q_e to help visualize the undefined transition in $B_{\mathbf{G}p}$ as illustrated in Fig. 2.5. Obviously, any run to the state q_e is a bad prefix that contradicts $\mathbf{G}p$.

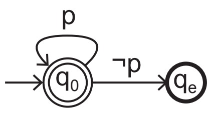

FIGURE 2.5: Visualize the undefined transition in $B_{\mathbf{G}p}$

B_f can also be used to determine whether f is liveness, or not. f is a liveness property, if B_f has no bad prefix but a constraint on the accepting states, i.e., from each accepting state there must exist a path back to itself. No bad prefix means that no finite run can contradict a liveness property. For any finite run that ends with an *undefined* transition, this finite run can always be extended to an infinite run. For a liveness property, all the undefined transitions (if any) in B_f are indeed redundant. Since from every state there is a path to an accepting state in B_f, a "good" thing can, of course, *eventually* happen.

E.g., every finite run that leads to the undefined transition in $B_{\mathbf{FG}p}$ can be extended to an infinite run, and this infinite run can eventually reach the accepting state q_1.

It is worth pointing out that there is a special *Büchi* automaton which accepts any path. In this sense, we say that this automaton is *universal*. On the contrary, if an automaton accepts no path, it is said to be *empty*. A set Q_i of states in the given *Büchi* automaton B is *universal*, if B becomes universal by redefining the set of initial states to be Q_i.

The automata mentioned above label the transitions with symbols (i.e., predicates). In contrast, we can also label the states of the automata with predicates. Thus, a *Kripke* structure, which defines a model M, can be seen as a state labeled automaton with all reachable states accepting.

For technical convenience, we define the *Büchi* automaton B_f of the given LTL property f as a state labeled automaton. Let AP_f be a set of atomic propositions derived from f. The *Büchi* automaton $B_f = (V_B, D_B, R_B, I_B, L_B, F_B)$ is defined in a way similar to the definition of the behavioral model M, where V_B and D_B are a set of variables and their (finite) domains, $R_B \subseteq D_B \times D_B$ is a transition relation, $I_B \subseteq D_B$ is an initial condition, $L_B : D_B \to 2^{AP_f}$ is a labeling function, and $F_B \subseteq D_B$ is an acceptance condition.

For example, the state labeled automata for $\mathbf{G}p$ and $\mathbf{FG}p$ are illustrated in Fig. 2.6.

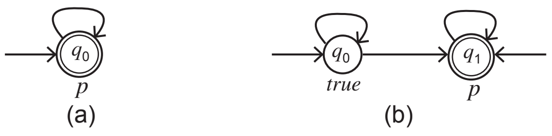

FIGURE 2.6: (a) $B_{\mathbf{G}p}$ and (b) $B_{\mathbf{FG}p}$

A run γ of B_f is an infinite sequence of states $\gamma = q_0, q_1, q_2, \cdots$ such that $q_0 \models I_B$ and $R_B(q_t, q_{t+1})$ for $t \geq 0$. Let $\mathit{inf}(\gamma)$ be a set of states that appear infinitely often in γ. A run γ is *accepting* if and only if $\mathit{inf}(\gamma) \cap \{q \mid q \models F_B\} \neq \varnothing$, i.e., at least one accepting state $q \models F_B$ appears in the ending loop of γ.

2.4 LTL Model Checking

Given a finite state model M and an LTL property f, LTL model checking [18] aims to answer the question: is it *true* that $M, \rho \models f$ for any initialized path ρ of M (i.e., ρ is an infinite path starting from some initial state of M)?

To solve this problem, we usually need to determine whether or not there exists an initialized path ρ in M such that $M, \rho \models \neg f$? i.e., the path ρ is a witness against the property f. Let $B_{\neg f}$ be a *Büchi* automaton generated from the negation of f. Then, we need to decide whether there exists an initialized path ρ that is accepted by $B_{\neg f}$.

Recall that the model M can be seen as a *Büchi* automaton with all the reachable states accepting. Let $\mathcal{L}(M)$ represents a set of (initialized) paths of M and $\mathcal{L}(B_{\neg f})$ a set of accepting runs of $B_{\neg f}$. If the intersection of $\mathcal{L}(M)$ and $\mathcal{L}(B_{\neg f})$ is *empty*, then f is satisfied with respect to M; otherwise, any path $\rho \in \mathcal{L}(M) \cap \mathcal{L}(B_{\neg f})$ is a counterexample. Therefore, we'd better determine whether $\mathcal{L}(M) \cap \mathcal{L}(B_{\neg f}) = \varnothing$ or not. The complexity of this process is PSPACE, i.e., polynomial in the size of the product of M and $B_{\neg f}$.

$\mathcal{L}(M) \cap \mathcal{L}(B_{\neg f})$ contains exactly all the accepting paths in the product of M and $B_{\neg f}$ that violate the property f. Generally, let $M = (V_M, D_M, R_M, I_M, L_M)$ over AP and $B_{\neg f} = (V_B, D_B, R_B, I_B, L_B, F_B)$ over $AP_{\neg f}$, then the product of M and $B_{\neg f}$ over $AP \cup AP_{\neg f}$ is a (*Büchi*) automaton denoted as $M \times B_{\neg f} = (V, D, R, I, L, F)$, where $V = V_M \cup V_B$, $D = D_M \times D_B$, $R = R_M \wedge R_B$, $I = I_M \wedge I_B$, $L : D \to 2^{AP \cup AP_{\neg f}}$ labels each compound state $(s, q) \in D_M \times D_B$ with the set of atomic propositions in $AP \cup AP_{\neg f}$ *true* in (s, q), and $F = F_B$ is acceptance condition.

An *infinite* path π of $M \times B_{\neg f}$ is a sequence of compound states $(s_0, q_0), (s_1, q_1), \cdots$ in $M \times B_{\neg f}$ such that

- $\rho = s_0, s_1, \cdots$ is an infinite *path* of M with $I_M(s_0)$ and $R_M(s_t, s_{t+1})$ for $t \geq 0$,
- $\gamma = q_0, q_1, \cdots$ is an infinite *run* of $B_{\neg f}$ with $I_B(q_0)$ and $R_B(q_t, q_{t+1})$ for $t \geq 0$,
- $L_M(s_t)$ is consistent with $L_B(q_t)$ for $t \geq 0$ with respect to the common propositions.

Let $\mathit{inf}(\pi)$ be the set of (compound) states that appear infinitely often in π. An infinite path π of $M \times B_{\neg f}$ is accepting if and only if $\mathit{inf}(\pi) \cap \{(s, q) \mid q \models F_B\} \neq \varnothing$.

LTL model checking algorithms can be implemented by exploring an *explicit state* space or a *symbolic state* space. As for explicit state space, we need to construct explicitly the state transition graph of $M \times B_{\neg f}$ as it is. In contrast, as for symbolic state space, the set of states and the set of transitions in $M \times B_{\neg f}$ are represented symbolically as *Boolean* formulas. This symbolic representation of states as well as of transitions can be implemented using Binary Decision Diagrams (BDDs) [41] or Conjunctive Normal Form (CNF). In the former case BDD-based tools, whereas in the latter case SAT solvers, are usually used to solve the LTL model checking problem.

2.5 Bounded Model Checking (BMC)

Symbolic state based model checking can handle efficiently a much larger state space than explicit state based model checking. However, the state space may grow exponentially in the number of the variables in the behavioral model under investigation. That is, doing model checking in a symbolic way does relieve the state space explosion problem to some extent, but still suffers from the state space explosion problem. Against this background, *bounded* model checking (BMC) is presented in [22], and later widely accepted in industry and academia.

Unlike traditional model checking, *bounded* model checking tries to search for counterexamples in the initialized paths of length bounded by some integer k. If no error is found, we increase the bound k until either an error is found, or some precomputed upper bound (Completeness Threshold) is reached, or until the problem becomes intractable. This method is *incomplete* if the completeness threshold is not reached.

In practice, BMC is usually encoded as a propositional satisfiability (SAT) problem. Given a model M and an LTL property f, the SAT problem is defined as follows:

$$|[M, f]|_k = |[M \times B_{\neg f}]|_k = (I_M(s_0) \wedge I_B(q_0)) \wedge (|[M]|_k \wedge |[B]|_k) \wedge |[C]|_k$$

where

- $I_M(s_0)$ and $I_B(q_0)$ are the initial conditions of M and $B_{\neg f}$ respectively,
- $|[M]|_k = \bigwedge_{i=1}^{k} R_M(s_{i-1}, s_i)$ and $|[B]|_k = \bigwedge_{i=1}^{k} R_B(q_{i-1}, q_i)$ encode the paths of length k in M and $B_{\neg f}$ respectively,
- the path constraint $|[C]|_k = \bigvee_{i=0}^{k} F(q_i)$ if F is a *finial* condition in case that f is a safety property, or $|[C]|_k = \bigvee_{l=0}^{k-1} ((s_l = s_k) \wedge (q_l = q_k) \wedge (\bigvee_{i=l}^{k-1} F(q_i)))$ if F is a *fairness* condition in case that f is a liveness property.

It is worth pointing out that in case of f being invariant, the SAT problem is simplified as $|[M, f]|_k = I_M(s_0) \wedge |[M]|_k \wedge |[C]|_k = I_M(s_0) \wedge (\bigwedge_{i=1}^{k} R_M(s_{i-1}, s_i)) \wedge (\bigvee_{i=0}^{k} F(s_i))$ with the final condition $F = \neg f$.

BMC can be implemented by explicit state-, BDD- and SAT- based search algorithms. The paper [42] compares the performance of these three search algorithms on 62 benchmarks drawn from commercial designs. The experimental results indicate that "BDD-based BMC is much faster" for finding *deep* counterexamples (of length, say, $k > 60$), while "SAT-based BMC is more effective than BDD-based BMC" for finding *shallow* counterexamples (of length, say, $k \leq 60$), but surprisingly "explicit state-based BMC (by means of random search) is comparably effective." As for the performance of SAT-based BMC, the experimental results [22] done at IBM, Intel and Compag show that "if k is small enough (typically not more than 60 to 80 cycles, depending on the model itself and the SAT solver), it outperforms BDD-based techniques." "The deeper the bug is (i.e., the longer the shortest path leading to it is), the less advantage (SAT-based) BMC has." In addition, the experimental results [43] indicate that "SAT solvers are quite effective in eliminating logic that is not relevant to a given property." In other words, "SAT solvers appear to have significant potential for identifying that set of variables once a suitable property is given." Of course, the performance of the BMC algorithms depends also strongly on the underlying hardware executing them as well as the complexity of the problems to be checked, and others.

In this thesis, we present an online model checking mechanism using the BMC technique to ensure the correctness of the actual execution trace within the next k (transition) steps starting from each (current) state monitored during the execution of the software system under investigation.

Chapter 3

Related Work

The work closely related to our online model checking are monitoring techniques in the sense that they need to observe the state information during system execution and then to check the correctness of the current execution trace of the target system. The concept of monitoring can date back to the assembly language era. Since then people have been using monitoring for testing, debugging, profiling, performance analysis, program optimization, and more. As a consequence, different monitoring techniques have been proposed since 1960. A comprehensive survey of program monitoring [28] was published in 1981, which examined the concepts, goals and limitations of the monitoring techniques at that time. The renewed interest in monitoring techniques nowadays is due to the increasing complexity of the software systems. A recent survey [44] in 2004 identified a wide spectrum of monitoring tools in the literature.

Generally, monitor observes the execution of the target system to sample the information of interest and then diagnoses, based on the collected data, whether the system behaves normally. The observed system may be a monolithic program, a parallel program, a distributed system, a real-time system, a hardware system, a network, or any combination thereof. The information of interest can be collected at various levels of abstraction, such as system level, process level, function level and statement level. The monitor can be implemented in the same process as the target system. In this case, the monitoring code is embedded in the target program. The monitor and the target system can also be implemented as separate processes on the same processor or on different processors. Monitoring is classified as *online*, if the collected data is analyzed while the target system is running; otherwise, it is classified as *offline*. Online monitoring is also called runtime monitoring. The interaction between the monitor and the target system is *synchronous*, if the target program must wait until the diagnosis is finished; *asynchronous*, if it does not need to wait for the result of the diagnosis.

In order to gain further insight into the (runtime) monitoring technique, some selected representative methods are described below in more detail.

Alamo

A Lightweight Architecture for Monitoring (Alamo) [45] is developed for monitoring C and Icon programs against safety properties. Prior to compilation, Alamo employs automatic program instrumentation to identify monitoring points and insert events into the source code of the target program. Typical events are memory references, heap allocations, procedure calls, I/O operations, and others. Control switches between the Execution Monitor (EM) and the Target Program (TP). The EM sends *event request* for desired events and transfers control to the TP. The TP executes until a desired event occurs, reports the event and transfers control back to the EM. Upon receiving the *event report*, the EM goes to check a predicate related to the event. The EM and the TP are implemented as *coroutine*s executing within the same address space. Therefore, the EM is allowed to inspect the state of the TP for additional information, such as the values of the variables, if necessary.

Anna Consistency Checking System

Anna (Annotated Ada) is an Ada extension for specifying constraints (Boolean expressions) as formal comments on Ada constructs, such as type, object, statement, exception, and others. Given an Ada program with annotations (formal comments), the Anna Consistency Checking System [46] transforms the annotations into checking functions and inserts into the Ada program the calls to these functions at the points of potential specification violation, e.g., assignments, procedure calls and type conversions. To improve the performance of the self-checking program in a multiprocessor system, for each checking function, a buffer task and a checking task are introduced, which execute concurrently with the Ada program being checked. The buffer task maintains a queue of check requests. The checking function enqueues check requests with data to the buffer task. The checking task dequeues them and performs the consistency checks. Upon detection of an inconsistency, the checking task can ignore the inconsistency, report the inconsistency to the Ada program, or terminate the Ada program.

BEE++

BEE++ [47] is an object-oriented application framework for the development of distributed dynamic analysis tools. The distributed program being monitored is instrumented a priori with sensors. A sensor provides a placeholder for an event that is either user-defined or predefined by BEE++. Whenever a sensor is encountered during program execution, an event (loaded with runtime data) is generated and sent off to one or more analysis tools bound to that sensor. These tools are used individually or in concert to detect the desired correctness or performance of the program. BEE++ provides a symmetric communication model similar to client-server approach, while allowing the client (the target program) and the server (the analysis tool) to be placed in the same entity, thereby providing peer-to-peer functionality. Events flow between the target program(s) and the analysis tool(s) over two distinct communication pathways: *Firehose* and *Trickle*. Firehose is used for high bandwidth communication from the target program(s) to the analysis tool(s) through an *event collection buffer*. Trickle is designed for asynchronous target program control and asynchronous monitoring in a way similar to the *ptrace* facility in Unix to control the thread/process and read/write data. BEE++ supports a variety of system architectures ranging from single processes to parallel and networked programs. Multiple clients (executing either as separate threads or processes) are able to connect to a single analysis tool and a single clients is able to connect to multiple analysis tools (running either on the same node or on different nodes). In the latter case, the client sends the events just to one analysis tool which in turn forwards the events to the other tools.

Observer-Worker System

An observer-worker system [48] aims to online check the behaviors of a distributed system in operation. It consists of two distinct components: a worker and an observer. The worker is an actual implementation of the system behaviors based on the given system specification. The observer is a formal model of some adequately selected aspects of the system behaviors that should be observed. The formal model is derived from the system specification, and thus can be used as observer for different implementations of the same system specification. The complexity of the formal model is restricted so as to guarantee that the correctness of the formal model can be verified exhaustively. In this sense, the observer can be seen as a reference, i.e., a correct implementation of the selected system behavior to be observed. The actual implementation (the worker) is continuously checked agains the reference (the observer) by comparing the worker behaviors with the observer behaviors at some observable output level. For this purpose, the observer need to know and to access the runtime information of the worker behavior. There are two ways in doing so: a) the worker informs explicitly the observer whenever an event of interest occurs; b) the knowledge of the worker behavior is directly accessible to the observer.

DynaMICs

Dynamic Monitoring with Integrity Constraints (DynaMICs) [49] is a software tool that supports the generation of constraints, the construction and insertion of constraint-checking code, and the tracing of failures with respect to requirements. A constraint is specified as event-condition-action. The event defines what and when the variables of interest need be monitored as well as when the constraint should be checked. The condition defines in first order logic the relationships between program variables, assertions on individual variables, and others. The action defines the response to a constraint violation, such as recording state in a history log, saving state for error recovery, performing state rollback, or initiating graceful degradation. For each constraint there is a set of instrumentation points in the program code at which the constraint-checking code is executed. The constraint-checking code may be an inline sequence of instructions, a function call, or a trigger that initiates the constraint check on a separate process. DynaMICs provides analysis tools to identify automatically the instrumentation points at the source code, intermediate code, or object code level. The work of the monitor is delegated either to the process executing the program code or to another process not necessarily on the same processor.

Falcon

Falcon [50] is a set of tools that support online capture of the application-level information, online analysis of the captured program information and online steering of the parallel program under investigation, which result in online modification of the program's execution. The information to be monitored ranges from single program variables to program (compound) states constituted of multiple program components running in parallel. The target program is instrumented a priori with the sensors and actuators generated from the given monitoring specification. Sensors are used to obtain the state information during program execution; whereas actuators to modify the execution of the target program. A single central monitor resides on a remote machine. Multiple local monitors execute on the target program's machine so that they are able to rapidly interact with the program. However, they may also run concurrently on different processors, using a buffer-based mechanism for communication between the target program and the monitoring threads. A local monitor can also inspect the program variables asynchronously with the execution of the target program by employing probe code without requiring prior instrumentation of the target program. A steering client runs as a separate program on a remote machine. It provides an interface for the user to interact with the target application. Several steering servers operate as threads in the application's address space, thereby gaining direct access to the application components and the ability to execute asynchronously with application threads. They read incoming monitoring events from the local monitors and respond to these events with appropriate steering actions. Each steering server shares with each local monitor a circular buffer located in jointly accessible memory. A steering server can perform simple program changes by enabling probe code, or perform more complex changes by enacting the actuators embedded in the application code.

Jass

Java with Assertions (Jass) [51] is a pre-compiler for Java programs annotated with assertions. The assertions are special formatted comments instrumented into the target Java program. Jass translates the assertions into Java code so that they can be checked during program execution. The assertions are defined as *boolean* expressions of Java extended with certain keywords as well as existential and universal quantifier over finite sets. Besides the usual assertions, such as method pre- and postconditions, class invariants, loop invariants and variants, and the like, Jass additionally supports refinement checks and trace assertions. Refinement checks are used to test whether a subclass is a behavioral subtype of its superclass. Trace assertions are used to monitor whether the trace of actual method invocations is valid.

JPaX

Java PathExplorer (JPax) [52] is a general-purpose monitoring mechanism, which can be easily extended to other programming languages. JPaX extracts the events of interest from the execution trace of the target program and then analyzes these events via a remote observer process, which may run on a different processor. JPaX consists of three main modules: an instrumentation module, an interconnection module, and an observer module. The instrumentation module instruments the target program using the given instrumentation script. During runtime the instrumented program emits the events of interest to the interconnection module. The interconnection module transmits these events further to the observer module. Upon receiving these events, the observer module dispatches them to a set of observer rules, each of which performs a particular analysis. The observer module currently provides two kinds of analysis: logic-based monitoring and error pattern analysis. The former checks the execution trace against the properties written in high level logics, such as safety properties and bounded liveness properties in linear temporal logic; the latter analyzes the execution trace using various error pattern detection algorithms.

MaC

Monitoring and Checking (MaC) [53] is an integrated framework for monitoring real-time systems, which can check general requirements related to an execution trace and to numerical computation. MaC consists of three components: a filter, an event recognizer, and a runtime checker. The filter is a collection of code fragments that can extract state information of the target system, such as the values of the variables and the function calls, and then send it to the event recognizer. These code fragments will be inserted into the implementation of the target system at the source code level or at the executable code level. From the received state information the event recognizer tries to detect the occurrence of an event defined at the requirements level in the given monitoring script, and then sends the detected event to the runtime checker. In addition, the event recognizer may also forward the values of the variables of interest to the runtime checker. Based on the events (and the values) it received thus far, the runtime checker is able to check the conformance of the sequence of events to the specification of requirements as well as the correctness of the requirements related to numerical computation.

MoP

Monitoring-oriented Programming (MoP) [54] is a general monitoring architecture independent of any specific programming language as well as any specific monitoring logic. Each MoP tool specializes this architecture to support specific programming languages and specific property logics. MoP consists of modules of three levels: process controllers at the interface level, code generators at the language level, and logic engines at the logic level. The workflow of the architecture is from the process controllers through the code generators to the logic engines and then from the logic engines through the code generators back to the process controllers. The properties to be checked are expressed in some formal logic and inserted as annotations in the form of comments at various user selected places in the target program. The process controller takes the annotated program as input and extracts the formal specifications from the annotations as output. The code generator takes the formal specifications as input and transform them into the formulas in some intermediate format as output. The logic engine then takes the formulas as input and produces abstract pseudocode for checking the formulas as output. The pseudocode is target language independent, therefore, the logic engine can be reused for different target languages. The code generator now takes the pseudocode as input and translates it into code fragments of the target language as output. The process controller then takes the code fragments as input and generates the executable code of the monitored program. The code fragments can be embedded into the target program or implemented as a different process, potentially on a different machine. In the latter case, the target program is instrumented so as to transmit the events of interest to the monitoring process.

Noninterference Monitoring

Noninterference Monitoring [55] is a hardware monitoring system designed for testing and debugging real-time software systems without interfering with the execution of the target system. The monitoring mechanism is implemented by using an auxiliary hardware (MC68000 processor) connected with the internal buses of the target system. In the monitoring phase, the activities of the target system are recorded at the user-defined conditional breakpoints. The runtime information can be collected at three abstraction levels: process level (e.g., system calls), function level (e.g., function calls) and instruction level (e.g., step-by-step execution trace). The target program is restricted to be written in a block-structured programming language in which a block is a function (or a procedure) and the scope of each variable is determined statically. The collected execution history of the target system is post-processed independent of the execution of the target system. The raw bus data is recorded in machine-level code, which contains not only the key values of the events of interest but also some redundant information. In the post-processing phase, the collected data is reorganized into meaningful information so as to represent the execution history in higher level logical views, e.g., process precedence graph and function calling tree.

Sentry System

The Sentry System [56] is a low precision and low cost monitoring system for sequential and concurrent C programs. A sentry is a monitoring program generated from the given target program with annotations. The annotations in the target programs are specially formatted comments derived from the properties to be checked. These comments are later replaced by the calls to macros for communication with the sentry. The sentry and the target program run in parallel and communicate with each other via shared memory. The target program is non-blocking in the sense that it never waits for the sentry. This means that some snapshots of the program state may be overwritten by the target program before being read by the sentry. In other words, some snapshots may be lost, hence, the precision is low. The sentry is able to check both safety and progress properties. Generally, the sentry reads a snapshot as it becomes available and then evaluates the properties. If a violation is detected, the sentry sends a signal to the target program, which may initiate some user-defined recovery action.

Temporal Rover

Temporal Rover [57] is a code generator. Given a program instrumented a priori with temporal properties as comments at some points in the source code, Temporal Rover parser converts this annotated program into an identical program except that the properties are now implemented in source code, too. This program code of the properties is compiled and linked as part of the program under investigation. During program execution, the correctness of the properties is checked by executing the generated code. Temporal Rover is able to deal with Linear Temporal Logic (LTL) and Metric Temporal Logic (MTL), an extension to LTL by supporting relative time and real-time constraints. In the case that a liveness property being checked keeps failed so far, Temporal Rover concludes conservatively that *so far* the property is failed, because it does not know during runtime whether or not the program will continue executing. The user is allowed to define reaction to the checking results. In addition, Temporal Rover has a special code generator targeted for embedded systems and concurrent systems. The generated verification code is allowed to be executed in a separate process or processor. The host and target code communicate via serial port, remote procedure call (RPC), or any other communication protocols.

Runtime Monitors for Distributed Hard Real-Time Systems

Validation of distributed systems needs to account for the interactions among nodes (or processes). Given a distributed hard real-time system under observation (SUO), which contains a fixed set of nodes and a fixed set of interconnects between nodes, the following three monitoring architectures at a conceptual level are presented in [12]:

1. (Single) Bus-Monitor Architecture: The monitor is attached to the data bus of the SUO, i.e., the monitor and the SUO share a common bus. The monitor receives messages over the bus just like any other process in the system and then does error checks based on the collected messages. If a violation is detected, the monitor will send messages to the other processes through the shared bus.
2. Single Process-Monitor Architecture: A dedicated monitor bus is introduced. Each process p_i in the SUO is attached to the data bus as well as the monitor bus. In addition, p_i is instrumented to send data to the monitor over the monitor bus. The monitor checks the correctness of the incoming data and signals the processes in the SUO if a violation is detected.
3. Distributed Process-Monitor Architecture: Each process p_i in the SUO has its own monitor M_i, which may be implemented on the same hardware as p_i. The distributed monitors are attached to a dedicated monitor bus for communicating with each other in order to reach agreement on diagnoses.

Online Failure Prediction

Recall that a failure is a kind of misbehaviors that can be observed from outside the system (see Section 2.1). Online failure prediction [26] aims to assess the potential occurrence of a failure in the near future in terms of seconds or minutes based on the measurements of the actual system parameters, such as resource usage, CPU load, system calls, etc., during runtime. There exists a wide spectrum of techniques dealing with online failure prediction in the literature. Almost 50 failure prediction methods have been surveyed in [26]. According to the type of the system parameters monitored at runtime, online failure prediction methods are classified into four categories as follows:

1. Failure Tracking: The occurrence of failures is tracked in terms of, say, the time of the occurrence and the types of the failures. This data can be analyzed to predict the potential failures that may come up in the near future. E.g., the probability distribution of the time to the next failure can be estimated based on the knowledge obtained from the previous failure occurrences. Due to sharing of resources, system failures may occur close together in a temporal as well as in a spatial sense.
2. Symptom Monitoring: An error inside the system may cause abnormal behaviors of the system parameters, such as memory usage, disk I/O, and unusual function calls. These side-effects are called symptoms of the error. By analyzing the system parameters monitored at runtime it is possible to detect symptoms that indicate

an upcoming failure. E.g., a functional relationship between the selected system parameters and the probability of failure occurrence can be established based on the previously recorded training data. By applying this function to the selected system parameters measured during runtime, it is possible to estimate the probability that a failure will occur. From a set of reference data points (i.e., training data) it is also possible to derive a decision boundary that partitions the data points into either failure-prone or non-failure-prone. Failure prediction can then be accomplished by checking on which side of the decision boundary the current date point is. In addition, failure prediction can also be performed by comparison of the currently measured value to the expected value computed from the system model with failure-free behaviors. If they differ significantly, an upcoming failure is predicted. Failure can also be predicted by analyzing several successive samples of the system parameters monitored during system operation.

3. Detected Error Reporting: When an error inside the system is detected, an error event is usually reported using some logging mechanism. The error reports that have occurred within some time interval (data window) before the current time can be analyzed so as to decide whether or not a failure will occur in the near future. E.g., from a set of event reports (training data) it is possible to identify some conditions or patterns that indicate the occurrence of failures. Based on the distribution of error types, the error generation rate, and the like, it is also possible to predict upcoming failures.
4. Undetected Error Auditing: Auditing searches for incorrect states (undetected errors) inside the system by checking on data that has or has not been used or produced. It can be applied offline as well as during runtime. E.g., memory auditing inspects data structures by checksumming. Failures can then be predicted based on the undetected errors found by auditing.

Quantitative Verification at Runtime

Quantitative verification [58] is an extension of conventional model checking to probabilistic models, which are typically variants of Markov chains, annotated with costs and rewards. The properties to be checked are expressed quantitatively in temporal logic extended with probabilistic and reward operators. Quantitative verification at runtime [59] deals with self-adaptive software, which is capable of adapting autonomously to changes in the environment. For this purpose, the probabilistic model of the software system is augmented with the parameters that reflect the changes in the environment in terms of, say, failure rates and costs/rewards. Whenever the changes in the environment are monitored, quantitative verification will be triggered to check whether or not the system model under the updated environment still satisfies the given quantitative properties. In case that a violation is detected, adaptive maintenance will be carried out. Quantitative verification can also help to make adaptive decisions so as to ensure that under the updated environment the system model after adaptation continues to satisfy the given quantitative properties.

Online Monitoring vs. Online Model Checking

Compared to offline verification techniques, such as testing, model checking, theorem proving, and the like, online monitoring is rather lightweight due to its concern with the correctness of the actual system execution[1] against the given properties, which makes it scale up well to deal with large complex systems. Generally speaking, online monitoring consists mainly of the following two parts:

- *Observer*: record the state information during system execution;
- *Analyzer*: analyze the collected data to figure out (or sometimes predict) anomaly in the target system.

Different monitoring techniques in the literature implement the observer as well as the analyzer in different ways. Online model checking does bear some similarity to online monitoring. It consists also of an observer and an analyzer. However, the implementation of the observer and the analyzer for online model checking is quite different from the existing monitoring techniques to our knowledge.

Given the source code of a target program and its behavioral model, we assume that a transition of the behavioral model corresponds to a predefined unit of execution of the target program. Here, a unit of execution is supposed to be atomic and thus determines the smallest distinguishable states of the program. It may be a statement or a composition of several statements within a basic block. A state is allowed to be monitored only before or after a smallest unit of execution of the target program.

The properties to be checked usually fall into two categories [28]: *state predicate* and *process predicate*. A state predicate is a *boolean* function defined on the state space of the target program, which is attached to a specific point of control in the program code. A process predicate is a *boolean* function defined on the set of sequences of program states, which is attached to a range of control points rather than a single point of control in the program text.

[1] i.e., a sequence of states monitored while the target system is running.

Online checking a state predicate is trivial for both online monitoring and online model checking due to its association with a specific point of control. For online monitoring, the state predicate is checked at the time when the program's execution reaches the related point of control. For online model checking, the state predicate may be checked before the program's execution reaches the related point of control (see Chapter 7).

Online checking a process predicate has to do with a sequence of program states. For a variable occurring in the property to be checked, the observer needs to probe every change in the value of this variable. It is important to decide the appropriate locations in the program code at which the program state should be monitored. The granularity of observation can thus have a large impact on online monitoring. If it is too coarse, important information may be missed; if it is too fine, the monitoring overhead will be too high. E.g., if the monitor tries to check for *overflow* after every arithmetic operation, it has to introduce more additional delay into the execution of the target system. In case that the property is derived from the system requirements, the *semantic gap* between the property and the execution trace make it usually difficult to establish a correct relationship between the low level state information and the high level (atomic) elements in the property.

Unlike online monitoring, online model checking does not directly check the correctness of the actual execution trace. Instead, a sequence of partial (behavioral) models of the target program that covers the actual execution trace is checked during runtime. The analyzer is thus implemented as online model checker. Errors found in some partial model may indicate errors in the actual execution trace or even predict errors that may happen in the future. To make online model checking available, each partial model covers the system behaviors up to a bounded length of k steps (for some appropriate positive integer k). Here, a step is a unit of execution of the program code, which corresponds to a transition in the behavioral model. By partitioning the program model into a finite set of k-bounded partial models, each state information observed at runtime is just used to decide which partial model should be checked next. Consequently, the observer needs to probe the state information at runtime more or less every k steps. The granularity of the observation is decided by the predefined bound k. This is different from what the observer of online monitoring does, which needs to monitor the values of the variables of interest whenever they are updated. In addition, the behavioral model is usually generated at a higher abstraction level than the source code of the target program. Compared to the source code, it is easier for the program model to establish a correct semantic relationship with the properties to be checked.

Of course, model checking is usually time consuming. Doing model checking online is a challenge. This is what the thesis tries to overcome.

Chapter 4

Online Model Checking Mechanism

No matter how fast a model checking algorithm does execute, it has to explore (explicitly or implicitly) all possible behaviors of the program model under investigation. This feature of exhaustive exploration makes the model checking process not only memory consuming but also time consuming, not to mention the state space explosion problem. It looks unrealistic and impossible to do model checking online while the target program is running. In this thesis, we don't mean to propose a faster model checking algorithm to ensure the universal correctness of the program model to be checked. Instead, we'd like to present an online model checking mechanism whereby efficient model checking techniques can be exploited during runtime to ensure the correctness of the actual execution trace of the target program. What's more, several acceleration techniques are also presented to speed up the online model checking process.

4.1 Online Model Checking

Briefly speaking, online model checking aims to ensure the correctness of the actual execution trace with respect to the given property by means of exploring during runtime a sequence of those (bounded) behavioral models that cover the execution trace of the target program.

4.1.1 Problem Statement

As a prerequisite for online model checking, the following information needs to be prepared in advance:

- *Source code* P of the software application to be online checked, which is written in a sequential programming language and instrumented with finitely many monitoring points so as to probe actual state information during program execution (see Chapter 5).
- *Behavioral Model* M of the software application, which is obtained in the software development phase or abstracted from the source code P (see Chapter 5). There may exist different behavioral models built at different levels of abstraction and/or from different perspectives, which reflect accordingly the behaviors of the software application at different levels of abstraction and/or from different perspectives.
- *Mapping function* $\alpha(s) = \widehat{s}$, which links each (concrete) state s of the program code P with the corresponding (abstract) state $\widehat{s}$ of the program model M if any, or else with a special state *null* if no appropriate abstract state is available. For different behavioral models of the same software application, there may exist respectively different mapping functions.
- *LTL Property* f, which specifies a characteristic predicate on the valid paths with respect to the given program model M.

For a software program under investigation, let P be the source code and M a behavioral model (at some level of abstraction and/or from some perspective) of P, then the mapping function $\alpha(s) = \widehat{s}$ from the state space of P to that of M is determined. Given any property f to be checked, online model checking aims to explore whether f holds along the execution trace of P during program execution by checking f against a sequence of (bounded) models derived from M following the program state monitored at runtime.

Errors in the behavioral model may indicate potential errors in the source code of the target program; therefore, instead of checking the correctness of the actual execution trace itself with respect to the property f, the basic idea of the online model checking is to check f against a set of bounded models that covers the actual execution trace of P.

Executing P results in a sequence of (possibly infinite) program states, called execution trace. A (program) state consists of a point in the control flow of P together with an assignment of values to all the variables of P at this point of control. The state space of P is a Cartesian product of the definition domains of all the components that constitute a state. Not all the states in the state space of P can be reached by executing P, regardless of the input values. It is intuitive to think of an execution trace of P as a trajectory of a point moving through space [28].

Fig. 4.1 below illustrates the three possible relationships between the behavioral model and the source code (implementation) of the target program:

(i) *equivalent* – each concrete path ρ corresponds to an abstract path $\widehat{\rho}$ and vice versa;

(ii) *over-approximate* – each concrete path ρ corresponds to an abstract path $\widehat{\rho}$, but *not* the other way round, i.e., the state space of the model is larger in this case;

(iii) *under-approximate* – each abstract path $\widehat{\rho}$ corresponds to a concrete path ρ, but *not* the other way round, i.e., the state space of the model is smaller in this case.

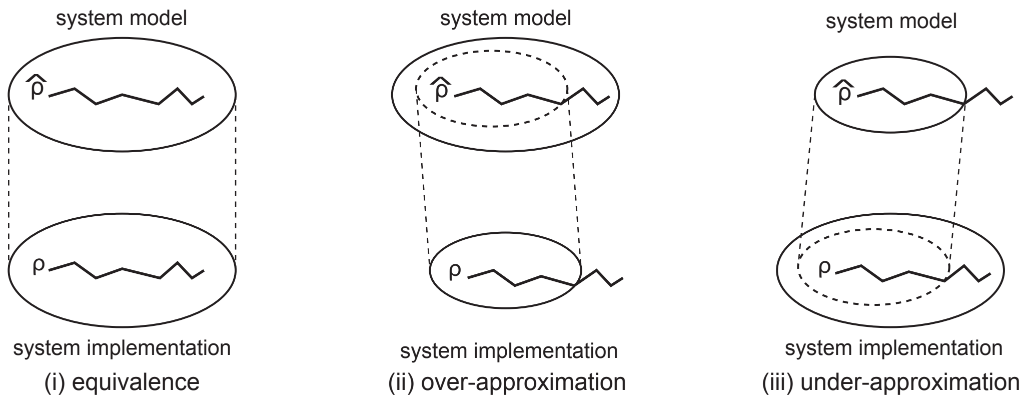

FIGURE 4.1: Three relationships between system model and system implementation

It is easy to reason that in the case of over-approximation, if no error is found in the behavioral model, there is no error in the program code, too. This checking result is a *true positive*. However, if an error path $\widehat{\rho}$ is found in the model, there may not really exist a corresponding error path ρ in the program code as depicted in Fig. 4.1 (ii), i.e., the error path $\widehat{\rho}$ is spurious. This checking result is a *false negative*. In the case of under-approximation, if an error path $\widehat{\rho}$ is found in the model, there must exist a corresponding error path ρ in the program code. This checking result is a *true negative*. However, if no error is found, there may still exist errors in the program code, because not all of the concrete paths, say the error path ρ, are reflected in state space of the model as depicted in Fig. 4.1 (iii). This checking result is a *false positive*.

Since we check the (behavioral) model instead of the actual execution trace of the target program, it is possible to deal with not only safety properties but also liveness properties. As mentioned in Section 2.3, a general LTL property can be decomposed into a safety property and a liveness property whose conjunction is the original [30]. It is fair to say that a nontrivial LTL property is either safety or liveness or a conjunction of a safety property and a liveness property. Without loss of generality, let $f = x \wedge y$ where x is a safety property and y a liveness property. Then, we have $\neg f = \neg x \vee \neg y$, which means any counterexample against f is either a *finite* witness against the safety property x or an *infinite* witness against the liveness property y.

Therefore, it is sufficient for us to concern ourselves with online model checking for safety properties as well as for liveness properties respectively.

4.1.2 Online Model Checking for Safety Properties

Given a behavioral model M and a safety property f to be checked, let $B_{\neg f}$ be the *Büchi* automaton generated from $\neg f$. Obviously, $B_{\neg f}$ accepts exactly those paths that contradict f. Recall that a finite bad prefix is sufficient to contradict a safety property (see Section 2.3). Consequently, it is possible to reduce the safety checking problem to the invariant checking problem, which can be solved by reachability analysis. However, if $B_{\neg f}$ is *nondeterministic*, which is usually the case, it is not that easy to decide whether a finite prefix is a bad prefix or not, especially when f contains some redundancy. One possible solution is to build a deterministic automaton $B'_{\neg f}$ from $B_{\neg f}$ by means of the subset construction [40]. The set of accepting states of $B'_{\neg f}$ is set to be those states that are universal. Now we need to simply check the *invariant* that the product of M and $B'_{\neg f}$ never reaches an accepting state of $B'_{\neg f}$.

Note, however, that $B'_{\neg f}$ is, in the worst case, exponential in the size of $B_{\neg f}$, and thus doubly exponential in the size of f, i.e., the number of subformulas of f. What's more, not every nondeterministic automaton can be transformed to an equivalent deterministic one. E.g., the LTL formula $\mathbf{FG}p$, no deterministic automaton can accept all those paths that satisfy $\mathbf{FG}p$ [30].

According to [40], a safety property f may fall into one of the following three types:

(i) *intentionally* safe – *all* the bad prefixes against f are informative. E.g., $\mathbf{G}p$, all its bad prefixes are informative in the sense that they reflect the whole reason why $\mathbf{G}p$ is violated.

(ii) *accidentally* safe – *not all* the bad prefixes against f are informative, but every computation that violates f has at least one informative bad prefix. E.g., $\mathbf{G}(p \vee (\mathbf{X}q \wedge \mathbf{X}\neg q)) = \mathbf{G}p$, the minimal bad prefixes against $\mathbf{G}p$ are also the bad prefixes of $\mathbf{G}(p \vee (\mathbf{X}q \wedge \mathbf{X}\neg q))$, but they do not reflect the whole reason why $\mathbf{G}(p \vee (\mathbf{X}q \wedge \mathbf{X}\neg q))$ is violated.

(iii) *pathologically* safe – there exists at least one computation that violates f but has no informative bad prefix. E.g., $(\mathbf{G}(q \vee \mathbf{FG}p) \wedge \mathbf{G}(r \vee \mathbf{FG}\neg p)) \vee \mathbf{G}q \vee \mathbf{G}r$, from its negation $(\mathbf{F}(\neg q \wedge \mathbf{GF}\neg p) \vee \mathbf{F}(\neg r \wedge \mathbf{GF}p)) \wedge \mathbf{F}\neg q \wedge \mathbf{F}\neg r = \mathbf{F}\neg q \wedge \mathbf{F}\neg r \wedge (\mathbf{GF}p \vee \mathbf{GF}\neg p)$, it is easy to see that its bad prefixes coincide with the bad prefixes of $\mathbf{G}q \vee \mathbf{G}r$. Due to the existence of the liveness subformulas $\mathbf{FG}p$ and $\mathbf{FG}\neg p$ in the formula, no bad prefixes of $\mathbf{G}q \vee \mathbf{G}r$, which is always finite, can tell the whole reason against $(\mathbf{G}(q \vee \mathbf{FG}p) \wedge \mathbf{G}(r \vee \mathbf{FG}\neg p)) \vee \mathbf{G}q \vee \mathbf{G}r$.

The safety properties that are accidentally safe or pathologically safe contain redundancy in them. E.g., the subformula $\mathbf{X}q \wedge \mathbf{X}\neg q = \mathbf{X}(q \wedge \neg q)$ is unsatisfiable, thus can be safely removed from $\mathbf{G}(p \vee (\mathbf{X}q \wedge \mathbf{X}\neg q))$; the subformula $\mathbf{GF}p \vee \mathbf{GF}\neg p$ is equivalent to $\mathbf{G}\mathit{true}$, thus can be safely removed from $\mathbf{F}\neg q \wedge \mathbf{F}\neg r \wedge (\mathbf{GF}p \vee \mathbf{GF}\neg p)$, too.

To simplify the problem, we assume that f contains no semantic redundancy or at least no syntactic redundancy. This assumption is reasonable and feasible. If a safety formula contains some liveness subformulas, these liveness subformulas are definitely redundant. We'd better optimize the given formula so that all the liveness subformulas are deleted. The resulting formula is thus syntactically safe in the sense that $\mathbf{X}$, $\mathbf{G}$ and $\mathbf{R}$ are the only temporal operators allowed. As a consequence, it is rather simple to figure out those states that are universal in the *Büchi* automaton $B_{\neg f}$, whereby there is no longer need to build the deterministic automaton from $B_{\neg f}$.

Let's redefine the acceptance condition of $B_{\neg f}$ as the set of all such states in $B_{\neg f}$ that are universal. Now we need to simply check the *invariant* that the product of M and $B_{\neg f}$ never reaches an accepting state of $B_{\neg f}$. In this sense, the new acceptance condition of $B_{\neg f}$ is also called *finial* condition, which defines a set of targets (or error states) with respect to f.

Of course, for the simple formula $f = \mathbf{G}p$, $B_{\neg f}$ is trivial and the acceptance condition is $\neg p$. We need to search only the state space of M for a finite path that reaches an error state satisfying $\neg p$. For a general safety property, we have to search the state space of $M \times B_{\neg f}$ for an error path in M. In the worst case, the whole state space of $M \times B_{\neg f}$ needs to be explored. If the state space of $M \times B_{\neg f}$ in terms of states and transitions is too large, it is impossible to conduct an exhaustive exploration within a reasonable time and memory consumption. In industry, for a complex software system, the state space of its behavioral model M alone may be too large to be searched exhaustively for checking a simple safety property like $\mathbf{G}p$. That is, even for a simple property, it is difficult to prove it completely (during the software development phase) [60]. This is one reason why we present the concept of online model checking.

By doing reachability analysis during runtime, we only need to search a smaller partial state space in $M \times B_{\neg f}$ that covers the actual execution trace of the target program in operation. The basic idea [61] is illustrated in Fig. 4.2. While the software system is running, whenever a monitoring point is reached, the current state s_i, i.e., the values of the system variables of interest, will be probed. The corresponding abstract state $\widehat{s_i} = \alpha(s_i)$ (if any) can thus be delivered to the online model checker. This in turn will trigger a new checking process: a partial state space starting from $\widehat{s_i}$ in $M \times B_{\neg f}$ will be explored within a predefined time limit (i.e., checking cycle) allocated to the online model checker in order to see whether there exists an error path from $\widehat{s_i}$ to a set of error states defined by the final condition of $B_{\neg f}$. In this way, the state space explosion problem can be avoided to a large extent.

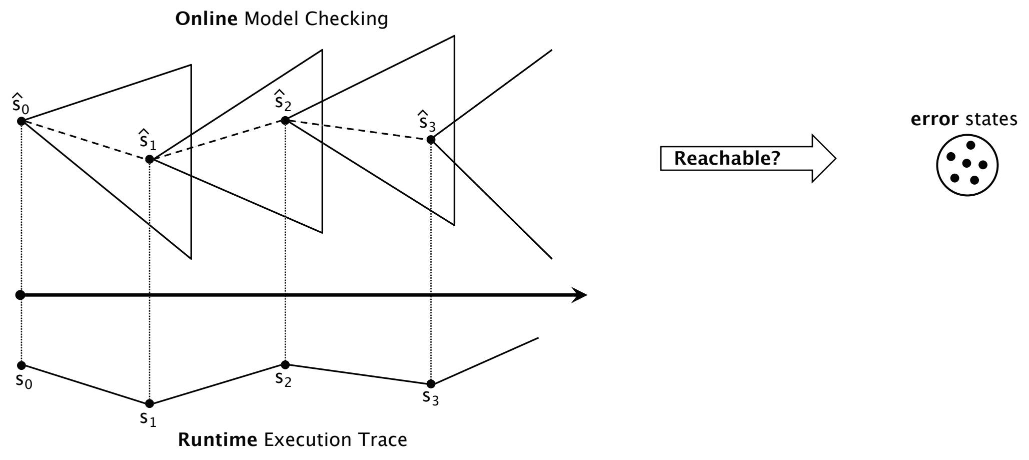

FIGURE 4.2: Online reachability checking

Due to the limited checking time, only finitely many (transition) steps in $M \times B_{\neg f}$, say, the next k steps starting from the monitored state can be explored in each checking cycle. Let $\mathcal{I}(\widehat{s_i})$ be the initial condition derived from the state $\widehat{s_i}$ monitored in the i'th checking cycle. The partial state space being explored in this checking cycle can thus be specified as $\mathcal{I}(\widehat{s_i}) \wedge |[M]|_k \wedge |[B]|_k$. Recall that $|[M]|_k = \bigwedge_{i=1}^{k} R_M(s_{i-1}, s_i)$ resp. $|[B]|_k = \bigwedge_{i=1}^{k} R_B(q_{i-1}, q_i)$ encodes the paths of length k in M resp. $B_{\neg f}$ (see Section 2.5). The path constraint $|[C]|_k = \bigvee_{i=0}^{k} F_B(q_i)$ tests if some state $q_i \models F_B$ within k steps. As a consequence, the online reachability problem in the i'th checking cycle can be formally defined as

$$|[M, f]|_k^i = |[M \times B_{\neg f}]|_k^i = \mathcal{I}(\widehat{s_i}) \wedge |[M]|_k \wedge |[B]|_k \wedge |[C]|_k.$$

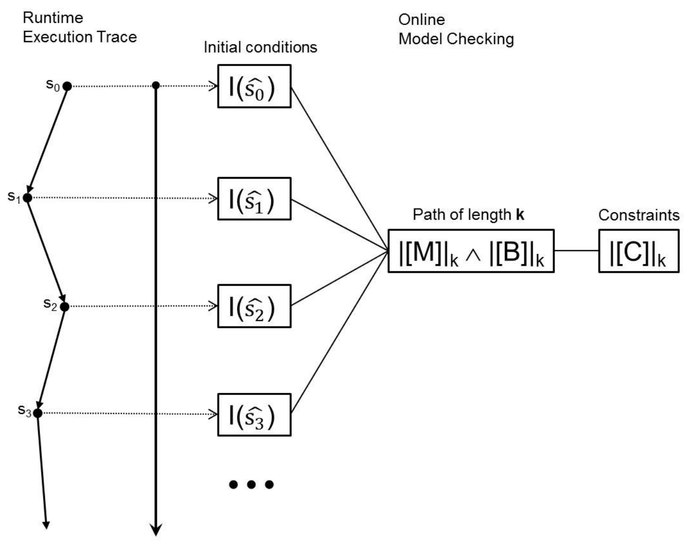

FIGURE 4.3: Bounded model checking during runtime

It is easy to see that online reachability checking is a kind of bounded model checking [22] applied during runtime as illustrated in Fig. 4.3.

Traditional bounded model checking is done *offline* in a way as illustrated in Fig. 4.4: starting from $k = 0$, if no error is found in any *initialized* path of length bounded by k, we progressively increase the bound k by 1, looking for errors in longer and longer traces, until either an error is found, or the complete threshold is reached, or until the checking problem becomes intractable. In our case, we leave the bound k unchanged all the way, instead, we change the initial condition $\mathcal{I}(\widehat{s_i})$ of the paths of length k in each checking cycle as illustrated in Fig 4.3. In effect, we are searching deeper and deeper in the state space of $M \times B_{\neg f}$ while the state space being checked in each cycle is relatively small. Therefore, by doing BMC during runtime it is quite possible to find deep, corner-case errors (if any) in the large state space of a highly sophisticated software system.

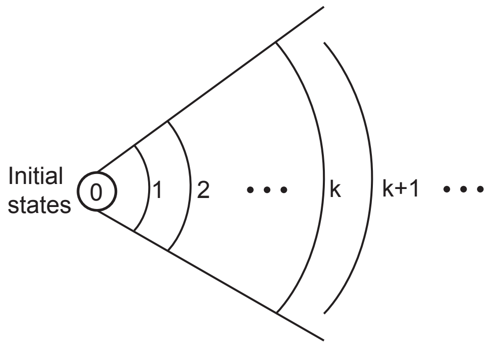

FIGURE 4.4: Traditional bounded model checking

In each checking cycle, the online model checker may return the following three possible checking results:

- *unsafe*: the checking process is finished in time, and an error path is found. However, in case that the system model is an over-approximation, the result may be a *false negative*. Anyway, the error path is useful for the user to figure out the reason.
- *safe*: the checking process is finished in time, but no error path is found. However, in case that the system model is an under-approximation, the result may be a *false positive*. Otherwise, the software system is *safe* within the next k steps relative to the current monitored state $\widehat{s_i}$.
- *unknown*: the checking process is enforcedly terminated due to timeout. In this case, the state space of the bounded model is not exhaustively explored. Since no error is found before timeout, it is reasonable to believe optimistically that there might be no error in the neighborhood of the currently monitored state.

4.1.3 Online Model Checking for Liveness Properties

Now let's consider the case that f is a liveness property. If we make every state in $B_{\neg f}$ accepting, then $B_{\neg f}$ becomes universal, i.e., it accepts any path. That is, there is no way to contradict a liveness property using any finite prefix. A run in $B_{\neg f}$ is accepting, if and only if it goes through some accepting state infinitely many times. For finite state automata, this means that we have to look for an accepting loop. A loop is accepting, if some state in it is accepting.

Every execution trace of a software system can be thought of as a trajectory of a point moving through space. It is not easy to detect a loop in it during runtime, because this requires to look backward into the "history" or look forward into the "future". Since we check the behavioral model, instead of the execution trace, of the software system, both the "history" and the "future" are accessible, theoretically speaking, it is possible for us to online check a liveness property during program execution.

Nevertheless, even on the model level it is still not easy to detect a loop during runtime, especially when the loop is too long to be detected in one checking cycle, e.g., its length is greater than the predefined bound k. Fortunately, we can avoid detecting a loop by means of the *state-recording translation* [62] from liveness checking to invariant checking, which can then be solved by online reachability analysis.

Recall that an infinite path $(s_0, s_1, \cdots, s_{i-1})(s_i, \cdots, s_n)$ with $s_n = s_i$ is usually made up of two parts: the finite prefix $(s_0, s_1, \cdots, s_{i-1})$ and the infinite loop $(s_i, \cdots, s_{n-1}, s_n)$. Let's call $(s_0, s_1, \cdots, s_{i-1})$ the *stem*, $(s_i, \cdots, s_{n-1})$ the *loop body*, and s_n (i.e., the second occurrence of s_i) the *loop closure*. The basic idea of the state-recording translation is to memorize the starting point s_i of every (potential) loop, a loop then is detected at some state s_n whenever $s_n = s_i$.

Given $M \times B_{\neg f} = (V, D, R, I, L, F)$ with $V = V_M \cup V_B$, $D = D_M \times D_B$, $R = R_M \wedge R_B$, $I = I_M \wedge I_B$, $L : D_M \times D_B \to 2^{AP \cup AP_{\neg f}}$, and $F = F_B$. In order to do the state-recording translation, three auxiliary variables are introduced: (i) $h \in D$ stores the starting state of a potential loop; (ii) $v_l \in \{st, lb, lc\}$ marks the location of a state in the path with st for stem, lb for $l$oop $b$ody, or lc for $l$oop $c$losure; (iii) $v_f \in \{true, false\}$ indicates the occurrence of an accepting state in the loop body. In addition, the symbol $\circledast$ is used to stand for an arbitrary but fixed state in D. The state-recording translation is then done by adding additional predicates to the initial condition and the transition relation of $M \times B_{\neg f}$ respectively. As a result, we can redefine the product of M and $B_{\neg f}$ as $M \times B_{\neg f} = (V', D', R', I', L', F')$, where

- $V' = V \cup \{v_l : D_l, v_f : D_f\}$ with $D_l = \{st, lb, lc\}$ and $D_f = \{true, false\}$;

- $D' = D \times D \times D_l \times D_f$;
- $R'((r, h, v_l, v_f), (r', h', v_l', v_f')) \equiv R(r, r') \wedge (R_1 \vee R_2 \vee R_3 \vee R_4 \vee R_5)$ with
  $R_1 \equiv (h = h') \wedge (h' = \circledast) \wedge (v_l = st) \wedge (v_l' = st) \wedge \neg v_f \wedge \neg v_f'$,
  $R_2 \equiv (h = \circledast) \wedge (h' = r') \wedge (v_l = st) \wedge (v_l' = lb) \wedge \neg v_f \wedge (v_f' \rightarrow r' \in F)$,
  $R_3 \equiv (h = h') \wedge (v_l = lb) \wedge (v_l' = lb) \wedge (v_f \rightarrow v_f') \wedge (v_f' \rightarrow v_f \vee r' \in F)$,
  $R_4 \equiv (h = h') \wedge (h' = r') \wedge (v_l = lb) \wedge (v_l' = lc) \wedge v_f \wedge v_f'$, and
  $R_5 \equiv (h = h') \wedge (v_l = lc) \wedge (v_l' = lc) \wedge v_f \wedge v_f')$;
- $I'(r, h, v_l, v_f) \equiv I(r) \wedge (I_1 \vee I_2)$ with
  $I_1 \equiv (h = \circledast) \wedge (v_l = st) \wedge \neg v_f$, and
  $I_2 \equiv (h = r) \wedge (v_l = lb) \wedge (v_f \rightarrow r \in F)$;
- $L'(r, h, v_l, v_f) = L(r)$;
- $F' \equiv (v_l = lc)$.

The initial condition I is partitioned into two categories characterized by the predicates I_1 and I_2 respectively. For each (old) initial state $r \in I$, r may be the starting point of some potential loop, or it may be not the starting point of any potential loop. The former case is governed by I_2, r is then saved in h, and $v_l = lb$, indicating that r is located in the body of a potential loop. If v_f is set to $true$, then r must be an accepting state; otherwise, it doesn't matter. The latter case is governed by I_1, r is not saved, h is thus set to the default value $\circledast$, and $v_l = st$, indicating that r is located on the *st*em of the path. In this case, r being accepting state or not is meaningless, therefore, $v_f = false$. Consequently, we may have three (new) initial states in I': $(r, h = \circledast, st, \neg v_f)$, $(r, h = r, lb, \neg v_f)$, and $(r, h = r, lb, v_f)$ for $r \in F$.

Generally, for each (old) state $r \in D$, let $x \neq r$ be any ancestor of r, then we may have the following seven types of new states in D': $(r, h = \circledast, st, \neg v_f)$, $(r, h = r, lb, \neg v_f)$, $(r, h = r, lb, v_f)$, $(r, h = x, lb, \neg v_f)$, $(r, h = x, lb, v_f)$, $(r, h = r, lc, v_f)$ and $(r, h = x, lc, v_f)$.

The transition relation R is partitioned into 5 categories characterized by the predicates R_1, R_2, R_3, R_4 and R_5 respectively. For each (old) transition $(r, r') \in R$, R_1 is applied to the state $(r, h = \circledast, st, \neg v_f)$ to get $(r', h = \circledast, st, \neg v_f)$, which are located on the stem of the path, i.e., no state is recorded; R_2 is also applied to the state $(r, h = \circledast, st, \neg v_f)$, which makes the state r' recorded, thus we have $(r', h = r', lb, \neg v_f)$ and $(r', h = r', lb, v_f)$ (for $r' \in F$) located in the body of a potential loop; R_3 is applied to the following four states $(r, h = r, lb, \neg v_f)$, $(r, h = r, lb, v_f)$, $(r, h = x, lb, \neg v_f)$, and $(r, h = x, lb, v_f)$ to produce $(r', h = r, lb, \neg v_f)$, $(r', h = r, lb, v_f)$, $(r', h = x, lb, \neg v_f)$ and $(r', h = x, lb, v_f)$ accordingly, which are all located in the loop body; R_4 is applied to the state $(r, h = r', lb, v_f)$ to obtain $(r', h = r', lc, v_f)$, which indicates that the previously saved state r' does occur again, thus, the potential loop is indeed a loop; R_5 is applied to two states $(r, h = r, lc, v_f)$ and $(r, h = x, lc, v_f)$ to produce $(r', h = r, lc, v_f)$ and $(r', h = x, lc, v_f)$ respectively.

Fig. 4.5 illustrates a state-recording process with respect to a path of $M \times B_{\neg f}$. From s_0 to s_{i-1} is the stem of the path, which is obtained by applying I_1 and then applying R_1 finitely many times. At the state $(s_{i-1}, \circledast, st, \neg v_f)$, applying R_2 we have $(s_i, s_i, lb, \neg v_f)$, in which the first occurrence of s_i is saved in h , indicating the starting point of a potential loop. Since s_i is not accepting, thus $v_f = \mathit{false}$. Afterwards, applying R_3 finitely many times until the state s_j is reached. Since s_j is an accepting state, thus v_f is changed from *false* to *true*, indicating that an accepting state has occurred in the loop body. Repeatedly applying R_3 thereafter until R_4 is applicable, which indicates that s_i is reached again. Consequently, we get the state (s_i, s_i, lc, v_f), i.e., the end of the loop. Thus, an accepting loop is detected. It is easy to see that the accepting loop is (synchronously) detected as soon as the saved state occurs the second time. Since then only R_5 is applicable, thus h, v_l and v_f remain unchanged.

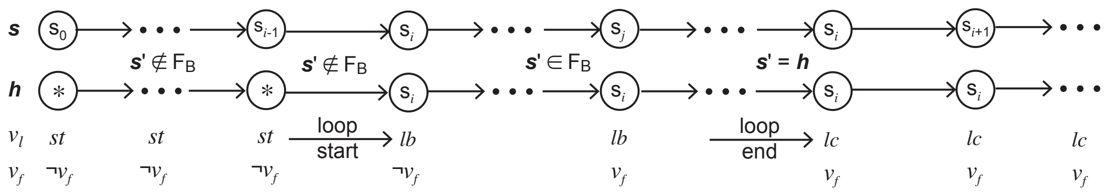

FIGURE 4.5: A state-recording process

The new definition of $M \times B_{\neg f}$ accepts a path whenever $v_l = lc$, which indicates that an accepting loop with respect to $B_{\neg f}$ is detected. As a consequence, the liveness checking problem is transformed into the invariant checking problem.

Let $|S|$ be the number of (reachable) states and $|T|$ the number of the transitions in the original $M \times B_{\neg f}$. After the state-recording translation is applied, the number of states is $\mathbf{O}(|S|^2)$ and the number of transitions is $\mathbf{O}(|S| \cdot |T|)$.

The above state-recording translation can be used to detect the shortest counterexample (if any). If it doesn't matter whether or not the counterexample is the shortest one, then we do not need to take any state as the starting point of a potential loop. Instead, we record only those accepting states as starting points. Thus, a loop is detected, whenever a saved accepting state occurs again. In this way, the auxiliary variable v_f is no longer useful. As a result, we need to redefine the initial condition and the transition relation of $M \times B_{\neg f}$ as follows:

- $I'(r, h, v_l, v_f) \equiv I(r) \wedge (I'_1 \vee I'_2)$ with
  $I'_1 \equiv (h = \circledast) \wedge (v_l = st)$
  $I'_2 \equiv (h = r) \wedge (v_l = lb) \wedge (r \in F)$

- $R'((r,h,v_l,v_f),(r',h',v_l',v_f')) \equiv R(r,r') \wedge (R_1' \vee R_2' \vee R_3' \vee R_4' \vee R_5')$ with
  $R_1' \equiv (h = h') \wedge (h' = \circledast) \wedge (v_l = st) \wedge (v_l' = st)$
  $R_2' \equiv (h = \circledast) \wedge (h' = r') \wedge (v_l = st) \wedge (v_l' = lb) \wedge (r' \in F)$
  $R_3' \equiv (h = h') \wedge (v_l = lb) \wedge (v_l' = lb)$
  $R_4' \equiv (h = h') \wedge (h' = r') \wedge (v_l = lb) \wedge (v_l' = lc)$
  $R_5' \equiv (h = h') \wedge (v_l = lc) \wedge (v_l' = lc)$

Let $|F|$ be the number of accepting states, which is usually smaller than $|S|$. Following this translation, the number of states is $\mathbf{O}(|S| \cdot |F|)$ in the worst case. In addition, the size of state is also smaller, since v_f is not used. Of course, there are also other ways to optimize the state-recording translation, but this is not the focus of this thesis.

4.1.4 Discussion

Before a software application is released (or deployed), it has usually already been verified or validated intensively by means of different checking techniques, e.g., static analysis, simulation and testing as well as model checking, and others. Therefore, it is reasonable to believe that the remaining errors (if any) may locate quite possible in some deeper corner in the state space of the software system. The online model checking mechanism is a relatively lightweight solution to detect such kind of subtle errors. We've addressed that the online model checking problem can be reduced to the online reachability checking problem. We now discuss in the general sense some points that should be noticed.

Falsification instead of Verification

Obviously, our online model checking is by nature a lightweight and incomplete method. It is suitable to falsify instead of verify the behavioral model against the given property. Due to its working on the model level, the online model checker is able to look into the near future in the system model. Thus, it can not only detect the error that has already happened, but also the error that has not happened yet, as illustrated in Fig. 4.6.

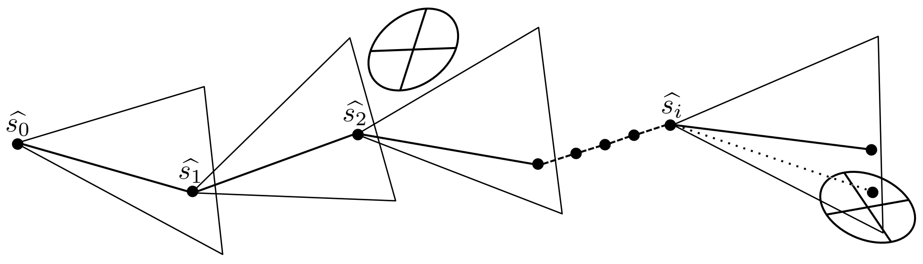

Figure 4.6: Online model checking process

If the partial state space starting from the current (abstract) state, say $\widehat{s_i}$, overlaps with an unsafe region marked as $\bigotimes$, the (potential) error can be detected by the online model checker even if the unsafe region is located much far away from the initial state(s), say $\widehat{s_0}$, of the system model. In case that the actual execution trace (see the dotted line) has already gone through the unsafe region, then the error has already happened. Otherwise, the error has not happened yet. In this sense, the online model checker is able to predict potential error(s) during system execution.

Let k be the predefined bound of the partial state space that needs to be explored from each monitored state and k' the number of steps that have been actually searched by the online model checker in each checking cycle. Then, an error located at the $(k'+1)$'th step can not be seen by the online model checker in the current checking cycle. E.g., the unsafe region near the state $\widehat{s_2}$ in Fig. 4.6 can not be seen by the online model checker in the second checking cycle.

Monitoring Points

There are a finite number of monitoring points. They are distributed in the source code of the software program to be checked such that

- any two adjacent monitoring points are at distance at most k steps[1];
- any location other than monitoring point in the source code is at distance at most k steps from some monitoring point;
- between any two locations with distance greater than k steps there must exist some monitoring point.

In Section 5.3 we'll discuss how to determine the monitoring points in the the control flow graph derived from the source code of the target program. For any two adjacent monitoring points x and y in the control flow graph, it is easy to reason that there may be more than one (loop free) path between x and y. Let $\delta_{max}(x, y)$ be the maximum distance between x and y. In addition to the above mentioned three conditions, we'd like to make $k - \delta_{max}(x, y) \leq \varepsilon$ as illustrated in Fig. 5.3, where $\varepsilon > 1$ is an integer far smaller than k. That is, for any monitoring point x, the succeeding monitoring point y is located at most k and at least $k - \varepsilon$ steps away from x.

No matter what property is to be checked, the state information is always monitored at each predefined monitoring point. That is, the locations of the monitoring points

[1]Here one step means one transition (step) in the model of the target program.

keep unchanged. There is no need to adjust the monitoring points in the source code for online checking different properties. That is, after the monitoring points have been instrumented in the source code, we are able to build the executable of the software program once and for all.

Pre-Checking and Post-Checking

In each checking cycle the online model checker just uses each (current) state s_i monitored during program execution to locate a starting point $\widehat{s_i} = \alpha(s_i)$ in state space of the program model (see Fig. 4.2). Thereafter, it conducts (semi-)exhaustive search for errors independent of the execution of the target program. Theoretically speaking, the online model checker may run ahead of or fall behind the progress of the target program.

We say that the online model checker is in the *pre*-checking mode, if it has explored at least k steps before timeout, i.e., $k \leq k'$; otherwise, it is in the *post*-checking mode, i.e., $k' < k$, as illustrated in Fig. 4.7, where the bullets represent the states monitored during system execution. It is easy to see that in the pre-checking mode the searched region has already covered the next monitored state before timeout, while in the post-checking mode the searched region may not cover the next monitored state, provided that the online model checker conducts an exhaustive search. Otherwise, the next monitored state may not be covered in both cases.

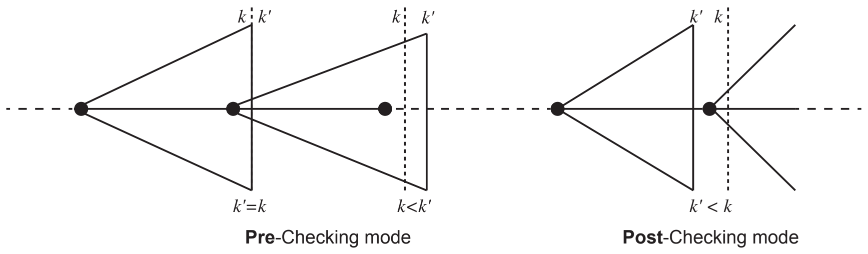

FIGURE 4.7: Pre-checking and Post-checking

For a nontrivial LTL formula f, we usually have to search in the state space of $M \times B_{\neg f}$. Given an execution trace of the program to be checked, let $s_0 \rightsquigarrow s_1 \rightsquigarrow \cdots \rightsquigarrow s_i \rightsquigarrow \cdots$ be the sequence of states monitored during runtime at the locations $l_0, l_1, \cdots, l_i, \cdots$ in the source code. For each state s_i, there is a *unique* state $\widehat{s_i} = \alpha(s_i)$ in M. For the sake of simplicity, we also use s_i to refer to $\widehat{s_i}$ in M in case of no ambiguity. Generally, s_i may be compatible with more than one state in $B_{\neg f}$. On the one hand, the automaton $B_{\neg f}$ is usually not deterministic; on the other hand, there may be more than one path going through the states $s_0, s_1, \cdots, s_i$ in M, which matches more than one run in $B_{\neg f}$. Let Q_i be the set of states in $B_{\neg f}$ such that for any state $q_i \in Q_i$, the compound state (s_i, q_i) is reachable via a path through the states $s_0, s_1, \cdots, s_i$ from some initial state in (s_0, q_0). Strictly speaking, in each checking cycle the online model checker should conduct the search in the partial state space of $M \times B_{\neg f}$ starting from the set of states (s_i, Q_i) at least k steps in depth.

Ideally, we assume that the online model checker conducts an exhaustive search. Let s_i be the state monitored at the location l_i for the current checking cycle and s_j be the state monitored at the location l_j for the next checking cycle. The starting points in the next checking cycle are then calculated based on the current starting states (s_i, Q_i) and the next monitored state s_j. Formally, let $\mathit{forwardReachableSet}(s_i, Q_i, k, \varepsilon)$ be the set of the (compound) states in $M \times B_{\neg f}$ that are *forward* reachable from (s_i, Q_i) within $[k - \varepsilon, k]$ steps in the current checking cycle.

In the pre-checking mode the online model checker can reach the states at least k steps in depth. Hence, we are able to obtain $\mathit{forwardReachableSet}(s_i, Q_i, k, \varepsilon)$ as a byproduct. Consequently, $(s_j, Q_j) = \{(s, q) \in \mathit{forwardsReachableSet}(s_i, Q_i, k, \varepsilon) \mid s = s_j\}$ are the starting points of the next checking cycle.

In the post-checking mode the online model checker may not reach the states up to k steps in depth before timeout. A possible solution is to resume the search work in the next checking cycle until $\mathit{forwardReachableSet}(s_i, Q_i, k, \varepsilon)$ is obtained. There is also a simple but not precise way to calculate (s_j, Q_j). That is, we can simply set Q_j to all the states in $B_{\neg f}$ that are compatible with s_j, i.e., $Q_j = \{q \in Q \mid s_j \models L_B(q)\}$. As a consequence, the states in (s_j, Q_j) are not ensured to be reachable from some initial state of $M \times B_{\neg f}$. Thus, the checking result may be *not* precise in case that some compound state (s_j, q_j) does reach an error state, but is not reachable from any initial state of $M \times B_{\neg f}$, i.e., the detected error is spurious.

In some special cases, there is no need to calculate (s_i, Q_i) for each monitored state s_i. E.g., the property $f = \mathbf{G}p$, the automaton $B_{\neg f}$ is trivial, therefore, the search is carried out in M; from the automaton $B_{\mathbf{FG}p}$ in Fig. 2.6 (b) it is easy to see that the states q_0 and q_1 are always reachable without any constraint on the paths to them, thus, we have $Q_i = \{q_0, q_1\}$ for each s_i.

Variables of Interest

In the case that there are too many variables in the behavioral model M, considering the monitoring overhead and the communication overhead between the target program and the online model checker, it is better to monitor only the most important variables in the system model, e.g., the program counter, the variables occurred in the property to be checked, and the variables in the cone of influence of the property, etc.. The user should decide what variables are of most interest in the system model.

Let $\widetilde{V}$ ($\subset V$) be the subset of the variables to be monitored during system execution. This incomplete information may not identify an individual state, but a set of states in the behavioral model, whose valuations on the variables in $\widetilde{V}$ equal to the observed ones. That is, there may exist more initial states in each checking cycle. This, in turn, means that the workload of the online reachability analysis may become heavy. One possible solution is to calculate an abstraction $\widetilde{M}$ of the system model based on the variables in $\widetilde{V}$. In this way, each monitored state can be mapped to a unique state in $\widetilde{M}$.

In this thesis, we assume that all the variables in M can be monitored during runtime.

Producer-Consumer Problem

During system execution a sequence of (concrete) states are probed whenever a monitoring point is reached. These states are stored in a (ring) buffer. In each checking cycle, the online model checker tries to take a state from the buffer as a new starting point, and then goes to search for an error path in the state space derived from the behavioral model and the property to be checked. This procedure is similar to the producer-consumer problem as shown in Fig. 4.8.

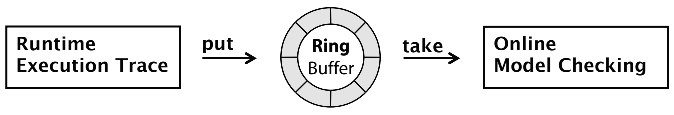

FIGURE 4.8: Producer-Consumer problem

In our case, we do not restrict the communication manner between the target system and the online model checker, be it synchronous or asynchronous. To reduce the impact of the online model checker on the execution of the target system, we do assume that the monitored states are put into the buffer without blocking. That is, if the buffer is *full*, the oldest state will be replaced by the latest one. On the other hand, if the buffer is *empty*, i.e., no state is available, the online model checker will resume the search work of the last checking cycle, provided that the work has not finished yet.

The buffer is used to balance the precessing speed of the both sides to some degree. Ideally, no state in the buffer would be dropped by the online model checker. In reality, it is not always the case. The producer might produce more data than the consumer could consume in time. Generally, the online model checking is carried out in the state space of $M \times B_{\neg f}$. Let s_i be the starting point of the last checking cycle, and s_j the starting point of the new checking cycle. If more than one state between s_i and s_j is dropped by the online model checker, then it is not easy to calculate precisely (s_j, Q_j), because s_j may be far away from s_i in the state space. In this case, the error detected may be spurious. Of course, this is no longer a problem for the invariant checking that is carried out in the state space of M, since every monitored state is reachable from some initial state of M.

In this thesis, if the search is carried out in the state space of $M \times B_{\neg f}$, we assume that no (monitored) state is dropped by the online model checker, i.e., the buffer is large enough to record all the states monitored during runtime.

Others

The user is allowed to check different properties during runtime, i.e., the properties to be checked are not necessary to be predefined in advance during the software development. The properties that have not been checked during the software development, or have been checked during the software development, but not completely proven, can be checked at runtime after the software is deployed.

The study [63] indicates that "many informal requirements are specified as properties of segments of program executions." In practice, it is usually not necessary to check the whole software program during runtime, instead, it is better to focus on some specific component or piece of code in the component, which is considered to be safety critical.

4.1.5 Prototype Implementation and Experimental Results

Given the behavioral model M and the property f, be it safety or liveness, theoretically, we are able to check M against f during runtime by means of online invariant checking. The invariant derived from M and f partitions the state space of $M \times B_{\neg f}$ into two non-overlapping regions: the set of valid states and the set of invalid states. What we need to do is to search for the potential error states in the partial state space of $M \times B_{\neg f}$ starting from each (abstract) state monitored during system execution.

As a proof of concept, we've implemented a prototype of the online model checking algorithm using the explicit state based breadth first search (BFS). We adopt the explicit state based instead of the symbolic state based search because by searching the explicit state space we are able to observe the internal structure of the system model and its influence on the performance of the online model checker. In addition, the implementation is relatively simple, it's a good starting point for us to learn intuitively the details of the online model checking mechanism.

Online Model Checking Algorithm for Safety Properties

Suppose that the time limit allocated to the online model checker is T time units. Each monitored (concrete) state is first mapped to the corresponding abstract state and then stored in the predefined ring *Buffer* during system execution. Algorithm 4.1 shows the pseudo code of the online model checking algorithm for safety properties. This algorithm is simplified in the sense that for each monitored state s the starting points for a new checking cycle are calculated in a simplified way: $(s, Q_s) = \{(s, q) \mid q \in Q \wedge s \models L_B(q)\}$, i.e., s is composed with any state q in $B_{\neg f}$ such that $s \models L_B(q)$.

The algorithm is straightforward: the online model checker waits until the *Buffer* is *not* empty, then it takes a state s from the *Buffer*. After calculating the starting points (s, Q_s), it then goes to search the partial state space starting from (s, Q_s) layer by layer until some error state is reached, or the whole partial state space has been explored, or the timeout occurs, whichever happens first. In the former two cases, the online model checking algorithm will terminate itself with the output *unsafe* or *safe* accordingly. In the latter case, the online model checker will output *safe*, if it has explored at least k steps in depth; or *unknown*, otherwise. Afterwards, a new checking cycle starts. The online model checker tries to take a new monitored state from the *Buffer*. If there is a state available, then it will repeat the above mentioned process; otherwise, it will resume the search work in the last checking cycle.

Concretely speaking, the algorithm is made up of the following three parts:

- **lines 3-9:** in the first checking cycle, the online model checker deals with the initial state s, which is stored in the *Buffer* initially. *r_set* is the set of starting (compound) states derived from s and $B_{\neg f}$ for the first checking cycle. If no error state is found in it, the states in *r_set* are pushed into the *queue* for further processing, followed by a *null* as delimiter between layers. The index of layer starts with zero. *step* records the index of the current layer being processed.

- **lines 12-25:** in the current checking cycle, the online model checker goes to explore the state space of $M \times B_{\neg f}$ layer by layer (by means of BFS). For each state r (other than *null*) in the *queue*, its successors are calculated. r is stored in the auxiliary queue *tmp_queue*. The states in $next(r)$ are then pushed into the *queue* for further processing in case that no error state is found in it. Otherwise, the online model checker will terminate itself with the output *unsafe*. Notice that the states of two adjacent layers in the unfolded state transition graph may coexist in the *queue*. The delimiter *null* is just used to separate them. When a *null* is encountered, it indicates that all the states in the current layer have been checked, therefore, *step* needs to be increased by one. It is worth pointing out that a state r' in $next(r)$ will not be added into the *queue* if $r' \in tmp\_queue \cup queue$, i.e., r' has already occurred in the current layer. That is, the number of states in *queue* may decrease to zero. In this case, the online model checker will terminate itself with the output *safe*. Otherwise, it will continue to check the states in the next layer until the timeout occurs.

- **lines 27-38:** when the timeout occurs, the online model checker will output *safe*, if at least k steps in depth has been explored; or *unknown*, otherwise. Thereafter, a new checking cycle starts. If the $Buffer$ is *not* empty, the online model checker takes a new current state s and then calculates a new set r_set of the starting points for the new checking cycle. Similar to the initial case, the states in r_set are pushed into the *queue* followed by *null*, if no error is found in it, and *step* is reset to 0. If no state is available in the $Buffer$, the online model checker will simply resume the search work left in the last checking cycle.

Algorithm 4.1: Explicit State-based Model Checking Algorithm

```
   input : $M, B, T, k$
   output : $safe, unsafe, unknown$
1  begin
2    wait ($isEmpty(Buffer) == false$) //wait until $Buffer \neq \emptyset$
3    set $timer$ to $T$ time units //initial checking cycle
4    $s \longleftarrow Buffer$ //pop
5    $r\_set = \{(s, q) \mid q \in I_B \wedge s \models L_B(q)\}$ //$s \in M$
6    if $\exists r \in r\_set$ s.t. $r.q \in F_B$ then return $unsafe$
7    $queue = r\_set$ //initialize queue
8    $queue \longleftarrow null$ //insert delimiter of layers
9    $step = 0$
10   $tmp\_queue = \emptyset$ //auxiliary queue
11   while $isEmpty(queue) == false$ do
12     $r = (s, q) \longleftarrow queue$ //dequeue
13     if $r \neq null$ then //$r$ is not delimiter of layer
14       $tmp\_queue \longleftarrow r$ //enqueue
15       $next(r) = \{(s', q') \mid s' \in next(s) \wedge q' \in next(q) \wedge s' \models L_B(q')\}$
16       for each $r' \in next(r)$ do
17         if $r'.q \in F_B$ then return $unsafe$
18         if $r' \notin tmp\_queue \cup queue$ then $queue \longleftarrow r'$ //enqueue
19       endfor
20     else //$null$ is dequeued (delimiter of layers)
21       $step{+}{+}$
22       $tmp\_queue = \emptyset$
23       if $isEmpty(qeueue) == false$ then $queue \longleftarrow null$ //delimiter
```

```
24          else return $safe$
25        endif
26        if $isTimeout(timer)$ then
27          if $step \geq k$ then output $safe$
28          else output $unknown$
29          set $timer$ to $T$ time units //new checking cycle
30          if $isEmpty(Buffer) == false$ then //$Buffer \neq \emptyset$
31            $s \longleftarrow Buffer$ //pop
32            $r\_set = \{(s,q) \mid q \in Q \wedge s \models L_B(q)\}$ //$s \in M$
33            if $\exists r \in r\_set$ $s.t.$ $r.q \in F_B$ then return $unsafe$
34            $queue = r\_set$ //initialize queue
35            $queue \longleftarrow null$ //insert delimiter of layer
36            $step = 0$
37            $tmp\_queue = \emptyset$
38          endif
39        endif
40      endwhile
41    end
```

The complexity of the algorithm is polynomial in the number of the states and transitions in the partial state space having been searched. In theory, compared to the simplified solution, it is usually more time-consuming to determine precisely the starting points for a new checking cycle. How to calculate the exact starting points efficiently is a topic worth further research. In practice, it should be acceptable to sacrifice accuracy for speed. Although the online model checker might report spurious errors, theoretically, it does not overlook real errors (if any) in the partial models having been checked.

Online Model Checking Algorithm for Liveness Properties

As for liveness checking, we have to search for an error loop in the state space of $M \times B_{\neg f}$. To this end, for each (compound) state $r = (s, q)$, we introduce a "memory" of r, denoted as $memo(r)$, to *memorize* those accepting states that can reach r. Initially, $memo(r) = \{r\}$, if q is an accepting state; or $memo(r) = \emptyset$, otherwise. We need then to modify the **for** loop in Algorithm 4.1 in the following way:

```
16      for each $r' \in next(r)$ do
17        if $r'.q \in F_B \wedge r' \in memo(r)$ then return $unsafe$
17'       $memo(r')$ += $memo(r)$ //update the memory of $r'$
18        if $r' \notin tmp\_queue \cup queue$ then $queue \longleftarrow r'$ //enqueue
19      endfor
```

It is easy to reason that an accepting loop (counterexample) can be detected, if r' is an accepting state and belongs to $memo(r)$, because r' is reachable from r and vice versa. In this case, the algorithm returns *unsafe*. Otherwise, $memo(r')$ is updated by inheriting the memory of r (line $17'$).

Experimental Results

Two experiments have been carried out on the Linux platform with Pentium-IV 3.00Ghz CPU and 1GB RAM. Our goal is to determine how far away the online model checker is able to look into the near future from each (monitored) state of the given model within a predefined time interval. The explicit state models are selected from the benchmark set BEEM [64]. They are finite state machines (FSM) derived from mutual exclusion algorithms, communications protocols, and so on, in research or industry settings. However, the corresponding source code is not available. This is not a problem because we focus on the model checking part, not on the state monitoring part. The real execution trace can be replaced by the "execution trace" generated randomly from the corresponding finite state machine. Thus, for each monitored state s_i, we have $\widehat{s_i} = s_i$.

Model	Type	State	Transition	Avg. Out-Degree	Max. Out-Degree	BFS Height	Max. Stack	Boolean Variables	Min. Look-ahead	Max. Look-ahead	Avg. Look-ahead
sorter_1	Controller	20544	30697	1.5	5	198	617	36	40	299	103
collision_1	Communications protocol	5593	10792	1.9	5	57	617	25	26	81	48.7
synapse_2	Protocol	61048	125334	2.1	18	41	2349	46	7	28	21.5
driving_phils_2	Mutual exclusion algorithm	33173	81854	2.5	9	150	3702	27	31	97	65.7
blocks_1	Planning and Scheduling	7057	18552	2.6	6	19	4263	23	8	21	14
peterson_1	Mutual exclusion algorithm	12498	33369	2.7	5	54	1862	30	13	39	31.7
szymanski_1	Mutual exclusion algorithm	20264	56701	2.8	3	72	2064	27	13	90	49.7
hanoi_1	Puzzle	6561	19680	3	3	256	4376	36	56	103	75.9
iprotocol_2	Communications protocol	29994	100489	3.4	7	91	443	39	18	451	50
phils_3	Mutual exclusion algorithm	729	2916	4	6	17	518	18	156	357	265
cyclic_scheduler_1	Protocol	4606	20480	4.4	8	55	1819	40	23	437	278
rushhour_1	Puzzle	1048	5446	5.2	9	73	535	28	66	248	150.7
rushhour_2	Puzzle	2242	12603	5.6	10	80	906	32	36	408	116.4
pouring_1	Puzzle	503	4481	8.9	9	13	348	16	42	101	71.9
reader_writer_2	Protocol	4104	49190	12	19	13	4097	25	4	16	9.9
pouring_2	Puzzle	51624	1232712	23.9	25	15	44509	18	1	4	2

TABLE 4.1: Experimental results of online invariant checking

One experiment is conducted for online invariant checking. 16 models are selected from the BEEM benchmark set. The features of these models are given in the number of states, the number of transitions, the average degrees of states, the height of BFS, and the maximal stack of DFS as well as the number of *Boolean* (state) variables. The invariant to be checked is $f = \mathbf{G}p$, where the *propositional* formula p is derived from the set of the states in each model. The experiment is designed to compute for each model how many steps (i.e., transitions) the online model checker is able to look ahead from each state in the model within $T = 1ms$[2]. The experimental results in Table 4.1 show the minimal, the maximal and the average look-ahead from the states of each model.

It is easy to see that the out-degrees of the states and the number of the *Boolean* variables have a large influence on the look-ahead performance of the online reachability checking.

The other experiment is conducted for online liveness checking. The selected model is *driving_phils_2*, which is derived from a mutual exclusion algorithm of processes accessing several resources, motivated by "The Driving Philosophers" [65]. The property to be checked is $f = \mathbf{G}(ac_0 \rightarrow \mathbf{F}gr_0)$, where the proposition ac_0 denotes that process 0 requests a resource and the proposition gr_0 denotes that the resource is granted to process 0. In other words, if process 0 requests a resource, the resource will be granted to it eventually. This is a liveness property, its negation is $\mathbf{F}(ac_0 \wedge \mathbf{G}(\neg gr_0))$. The *Büchi* automaton $B_{\neg f}$ is illustrated in Fig. 4.9. It shows that the error path must end with a loop satisfying $\mathbf{G}(\neg gr_0)$, i.e., $\neg gr_0$ holds in each state on the loop.

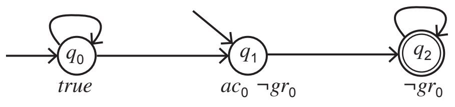

FIGURE 4.9: *Büchi* Automaton of $\neg f = \mathbf{F}(ac_0 \wedge \mathbf{G}(\neg gr_0))$

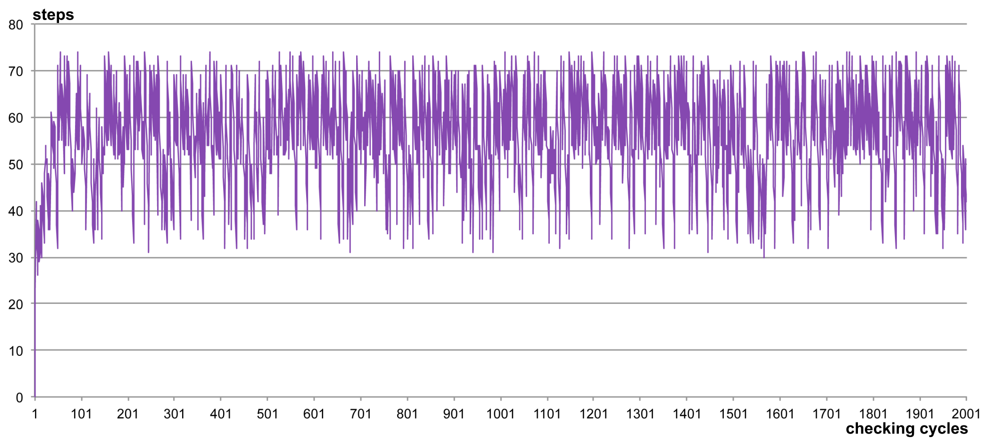

FIGURE 4.10: Experimental results of online liveness checking

In this experiment, the checking cycle is set to $T = 5ms$, i.e, the online model checker has at most $5ms$ to do the search work in each checking cycle. The monitored states are sampled every five steps from the execution trace generated randomly from the model. The experimental result in Fig. 4.10 is obtained by running 2000 checking cycles. The X-axis represents the checking cycles and the Y-axis (transition) steps checked in each checking cycle. The property is not violated at least up to these 2000 checking cycles. The minimal look-ahead is 23 steps, the maximal look-ahead is 74 steps and the average look-head is 57.2 steps relative to the corresponding monitored states.

[2]The actual runtime may be more than $1ms$ in case that the timeout signal is not processed in time.

Extend Online Model Checking Algorithm by means of Random Search

As far as explicit state based search is concerned, *random search* is a natural way to improve the performance of our online model checking algorithm in terms of the look-ahead steps (in each checking cycle) so that those models with a high density of states can also be checked efficiently during runtime.

The paper [42] also justifies the random search approach by saying that "since BMC is generally used to find counterexamples in contrast to proving that a property holds", and "the ability to use a random search is an advantage of the explicit state engine", although "this process may miss executions and thus counterexamples". The experimental results in [42] indicate that the explicit state based random search for BMC is surprisingly "as effective as SAT-based BMC in finding short counterexamples for safety properties". Here the "short" counterexamples are at a depth of at most 50 steps in the state space.

In our case, a simple solution is to conduct a random BFS by introducing a new function $select(r, d_{lim})$ to select randomly d_{lim} successors of r if necessary, where d_{lim} is a predefined threshold on the out-degrees of the states in the model. If $|next(r)| \leq d_{lim}$, $select(r, d_{lim}) = next(r)$. Otherwise, $|next(r)| > d_{lim}$, then $select(r, d_{lim})$ selects randomly or in some heuristic way d_{lim} states from $next(r)$. As a result, the **for** loop in Algorithm 4.1 can be modified in the following way:

```
16    for each $r' \in select(r, d_{lim})$ do
17      acceptance checking code
18      if $r' \notin tmp\_queue \cup queue$ then $queue \longleftarrow r'$ //enqueue
19    endfor
```

Experimental Results

Two experiments have been carried out on a Linux platform with Intel Core 2 Duo 3.00Ghz CPU and 4G RAM. The model used is *driving_phils_2*, which has 33,173 states and 81,854 transitions. The average out-degree of the model is 2.5, the maximal out-degree is 9 and the minimal one is 1. The state variables are encoded into 27 *Boolean* variables. The timer is set to $T = 5ms$ for each checking cycle. The experimental results in Fig. 4.11 and Fig. 4.12 are obtained by running 200 checking cycles respectively.

For online invariant checking, the minimal, maximal and average look-ahead are 63, 146 and 113.3 steps respectively in the case $d_{lim} = 2$; 48, 103, and 85.8 in the case $d_{lim} = 3$; 47, 102, and 83.0 in the case that no random search is used. The liveness property to be checked is also $\mathbf{G}(ac_0 \rightarrow \mathbf{F} gr_0)$. The minimal, maximal and average look-ahead are 44, 143, and 99.4 steps respectively in the case $d_{lim} = 2$; 23, 85 and 64.5 in the case

$d_{lim} = 3$; 30, 96 and 64.8 in the case that no random search is used. Considering that the average out-degree of the model is 2.5, we are not able to gain much improvement in performance by setting d_{lim} to 3. However, we do gain better performance by setting d_{lim} to 2.

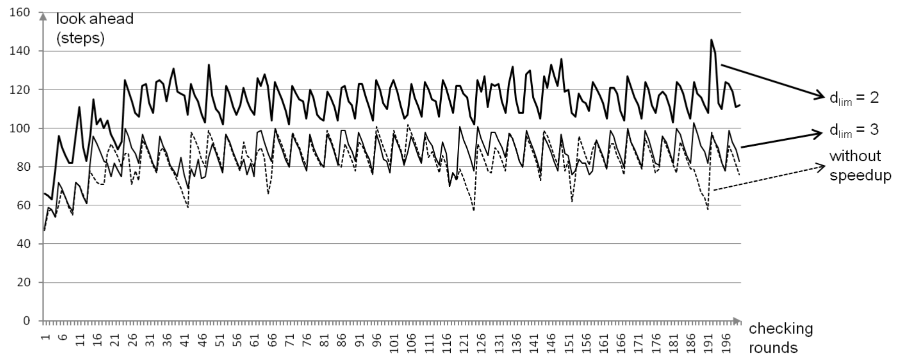

FIGURE 4.11: Online invariant checking by means of random BFS

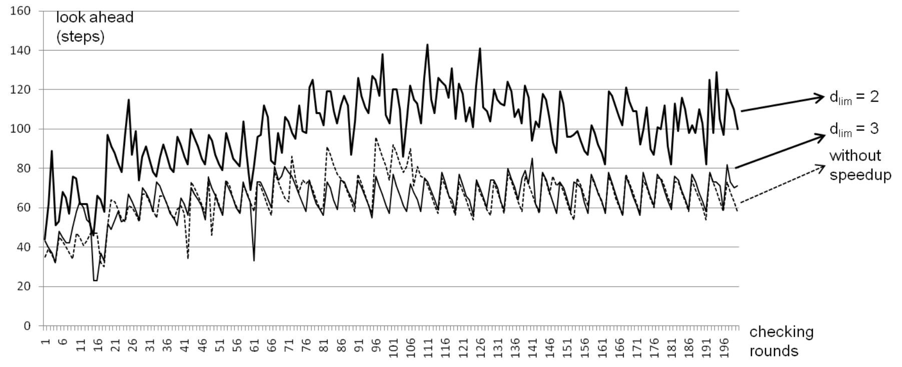

FIGURE 4.12: Online liveness checking by means of random BFS

In the prototype the function $select(r, d_{lim})$ is implemented to select states randomly. In practice, the source code is usually validated by simulation and testing in the software development phase. Some heuristic information learned in this process, e.g., which control branch is more important, can be used to weight the related transitions in the model. Accordingly, the function $select()$ can be implemented in such a way that important transitions have more chance to be selected. If a state is reached more than one time, at each time a well-defined $select()$ function should select different successors of this state. In addition, the threshold d_{lim} can also be adjusted dynamically. In effect, the function $select()$ samples randomly the bounded paths of the model during runtime.

4.2 Accelerating Online Model Checking

Doing model checking online has to suffer from the limited execution time allocated to each checking cycle. The performance of online model checking depends on the search algorithm, the underlying hardware and the operating system as well as the complexity of the problem to be checked. Therefore, we can accelerate online model checking by means of reducing the workload, speeding up the search engine and using parallel computing.

4.2.1 Reducing Workload

After the given model checking problem is transformed into the reachability problem, the product of M and $B_{\neg f}$ is reformulated as a finite automaton, denoted still as $M \times B_{\neg f} = (V, D, R, I, L, F)$ for the sake of convenience.

Offline Forward Exploration

The transition relation $R(r, r')$ defines a one-step transition relation in the state space of $M \times B_{\neg f}$. That is, applying R to any (compound) state $r \in D$, we get the one-step successors of r. Let $R^1 = R$, then we define the two-step transition relation $R^2(r, r'') = \exists r'.R^1(r, r') \wedge R^1(r', r'')$. Now applying R^2 to any state $r \in D$, we are able to get the two-step successors of r. Theoretically, starting from R^1 we are able to *offline* calculate multi-step transition relations $R^2, R^3, \cdots, R^k$ in the state space of $M \times B_{\neg f}$. Applying R^m for $1 \leq m \leq k$ to any state $r \in D$, we are able to get the m-step successors of r.

Recall that for safety properties there is no way to extend a bad prefix to an infinite run which is accepting (see Section 2.3). That is, once an error state is reached, the states that follow it are all identified as error states. As a consequence, it is safe to search for error states by multi-step jumping in the state space of $M \times B_{\neg f}$.

Let s be a state monitored at the location l for the current checking cycle. Upon receiving s, the online model checker first calculates a set *r_set* of the compound states compatible with s as starting states of this checking cycle, and then simply applies R^k to *r_set* to see whether or not the the k-step successors of the states in *r_set* may reach some error state. It is worth pointing out that if an error state is detected, inbetween s and the the error state there may exist other error states, i.e., the error state detected by multi-step jumping may be not the one closest to s.

Now let s' be a state monitored at the location l' for the next checking cycle. Notice that the distance $d(l, l')$ of any two adjacent monitoring points l and l' falls into the *integer* interval $[k-\varepsilon, k]$. Therefore, the set r'_set of the starting states in the next checking cycle is $r'\_set = \{r' \mid r' \in \mathit{forwardReachableSet}(s, Q, k, \varepsilon) \wedge r.s = s'\}$. Similarly, by applying R^k to r'_set, the online model checker is able to see whether or not the k-step successors of the states in r'_set may reach some error state. In this way, as illustrated in Fig. 4.13, the online model checker avoids searching step by step in the state space of $M \times B_{\neg f}$.

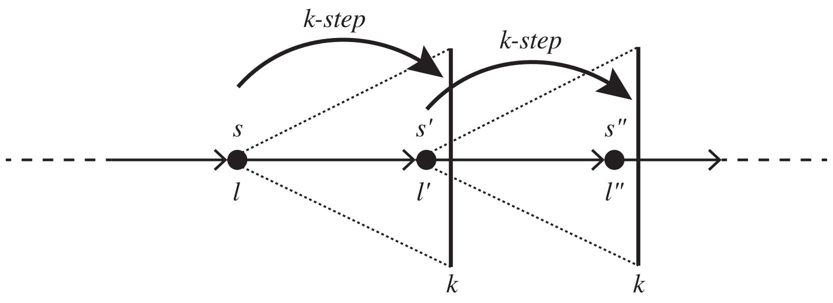

FIGURE 4.13: k-step forward jump

In practice, if R^k is not available due to the complexity of the behavioral model M, then applying some R^m for $1 < m < k$ can also speed up the online checking process.

Taking this offline forward exploration into account, the pre-checking and post-checking defined in Section 4.1.4 is no longer suitable. We say that the online model checker is in the pre-checking mode, if there is one and only one state available in the ring buffer; otherwise, it is in the post-checking mode, i.e., there is more than one state available.

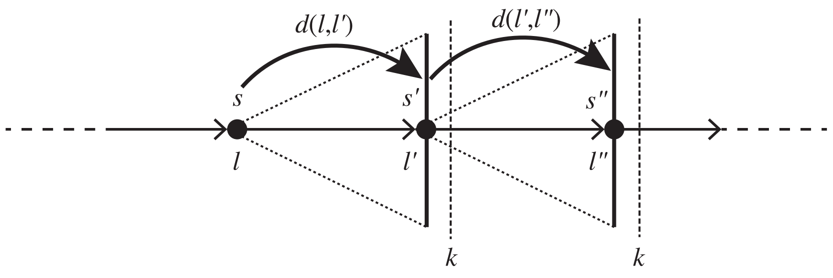

FIGURE 4.14: d-step forward jump

Although on the model level the distance $d(l, l')$ of any two adjacent monitoring points l and l' may be not unique, $d(l, l')$ $(\in [k-\varepsilon, k])$ is unique with respect to the given actual execution trace of the program to be checked. If this unique $d(l, l')$ is available during runtime, then we can also make the online model checker each time take from the buffer two states s and s' in a row whenever it is in the post-checking mode. Instead of jumping k steps forward, the online model checker in this case jumps $d(l, l')$ steps forward, provided that s is monitored at the location l and s' at the location l', as illustrated in Fig. 4.14. In this way, it is relatively easy to to calculate the exact starting points for the new checking cycle. Notice that $d(l, l')$ and $d(l', l'')$ may be not equal.

Offline Backward Exploration

The final acceptance condition F (of $M \times B_{\neg f}$) defines the set of error states. Let's extend the (initial) unsafe condition F to become $F^* = F_0 \vee F_1 \vee \cdots \vee F_n$ by offline backward exploration up to n steps starting from $F_0 = F$, where $F_i = \{r \mid R(r, r') \wedge r' \in F_{i-1}\}$ for $0 < i \leq n$, as illustrated in Fig. 4.15. In effect, the workload of the online model checker is reduced to a large degree by calculating in advance the backward reachable set F^* from the (original) set F of error states.

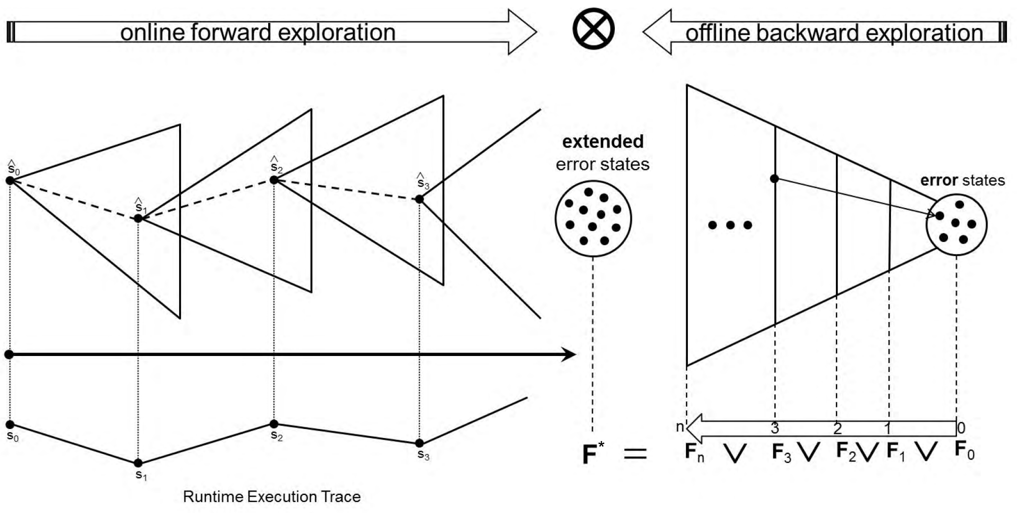

FIGURE 4.15: Speed up online model checking

Due to time and memory limits, it is usually difficult for the online model checker to explore forward too deep in the state space in each checking cycle. Therefore, the bound k of the online depth-limited search should be set to a relatively small number. However, the calculation of F^* is carried out *offline*, whereby time and memory are no longer a big problem. In addition, many existing efficient approaches to reachability analysis can be used to calculate F^*. As a consequence, it is possible to explore backward much deeper in the state space to be checked. Thus, it is reasonable to assume that n is much larger than k. Of course, this doesn't mean that n is large enough to solve the problem being checked. The calculation of F^* still suffers from the state space explosion problem, n is a limit obtained under a reasonable time and memory consumption.

Without the offline backward exploration, the online model checker is able to look ahead ideally k steps in each checking cycle; with the offline backward exploration, the online model checker is now able to look ahead ideally $k + n$ steps.

As for the liveness checking problem, although we can transform it into the safety checking problem by means of state-recording translation (see Section 4.1.3), the state space is increased quadratically in the number of states, not to mention the number of transitions. To make the online liveness checking process more efficient, we'd better make the computation during runtime *as simple as possible, but not simpler.*

Loop detection is the key to the liveness checking problem, which usually leads to additional time and memory consumption. A possible solution is to leave the loop detection done offline. The point is to calculate beforehand a new acceptance condition $F_0 \subseteq F$ such that for any (compound) state $(s_i, q_i) \models F_0$, there exists a loop of length w ($w \geq 1$) through (s_i, q_i) in $M \times B_{\neg f}$, i.e., $F_0 = \bigvee_{t=1}^{w} C_l^t$ with $C_l^t = \exists r_1, r_2, \cdots, r_t . F(r_0) \wedge R(r_0, r_1) \wedge R(r_1, r_2) \wedge \cdots \wedge R(r_{t-1}, r_t) \wedge (r_t = r_0)$. Then, as illustrated in Fig. 4.16, we need to check during runtime whether or not the states in F_0 is reachable.

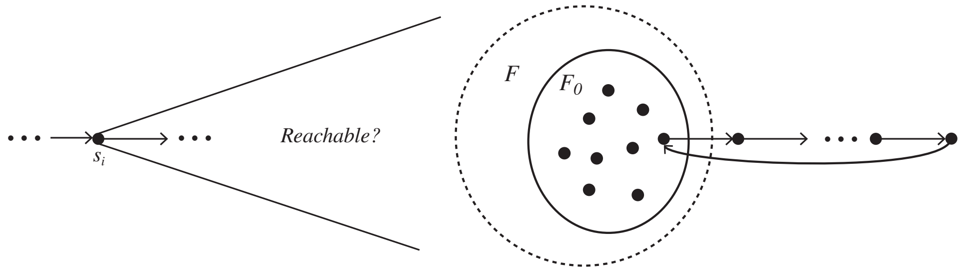

FIGURE 4.16: Speed up liveness checking

For a finite state system, the upper bound of w is the depth of backward breadth first search starting from the accepting states in $M \times B_{\neg f}$. In practice, it is usually difficult to reach this upper bound due to the state space explosion problem. Since F_0 is calculated offline, many existing model checking techniques can be exploited to calculate F_0. Thus, it is reasonable to assume that w is much larger than k.

In the case that F_0 can not be calculated precisely, let $\widetilde{F_0}$ ($\subset F_0$) be the partial solution obtained within a reasonable time and memory consumption. Then, $F' = F - \widetilde{F_0}$ is the set of accepting states *unknown* whether or not there exists a loop through them. We are able to apply the variant of the state-recording translation to $M \times B_{\neg f}$, whereby only the states in F' are selected as the starting points of the potential loops (see Section 4.1.3). In this way, the state space to be searched can be further reduced to some degree.

The focus of this thesis is on the online model checking mechanism and its applications. It is reasonable to assume that the multi-step transition relation R^m and/or the backward reachable set F^* (if any) are provided in advance.

4.2.2 Online Symbolic Model Checking

Although there still exists some room to optimize the explicit state based model checking algorithms, for many complex systems, it is not convenient to store and operate on the large state transition graph represented explicitly in terms of states and transitions due to the huge memory and time consumption. It is widely accepted that the symbolic state based checking methods are able to handle efficiently a much larger state space than the explicit state-based ones. By symbolic state based model checking, the states and the transitions between states are represented implicitly as formulas in quantified propositional logic (see Section 2.2).

Online Symbolic Model Checking Algorithm

Given R^m and F^*, for the sake of convenience, let $M \times B_{\neg f} = (V, D, R^m, I, L, F^*)$ denote the *finite* automaton obtained by the following *offline preprocessing*:

1. reduce the general model checking problem to the reachability checking problem;
2. calculate a multi-step transition relation R^m for some $m > 1$; and
3. extend the set F of error states to F^* by n-bounded backward reachability analysis.

As a consequence, we are able to conduct the online model checking by means of *jumping* in the state space of $M \times B_{\neg f}$, as described in Algorithm 4.2. The algorithm is mainly composed of the following three parts:

- **lines 3-7:** in the first checking cycle, the online model checker deals with the initial state s, which is stored in the *Buffer* initially. *r_set* is the starting (compound) states derived from s and $B_{\neg f}$ for the first checking cycle. If *r_set* contains error state(s), then the online model checker will terminate itself with the output *unsafe*. *jump* records the number of jumps made by the search algorithm.
- **lines 9-12:** in the current checking cycle, *r′_set* is the set of m-step successors of the set *r_set* of (current) states. If no error state is found in it, *r_set* is reset to *r′_set*. The online model checker will continue to jump another m steps until the timeout occurs.
- **lines 14-21:** when the timeout occurs, the online model checker will output *safe*, if at least k steps in depth has been searched; or *unknown*, otherwise. Thereafter, a new checking cycle starts. If the *Buffer* is *not* empty, the online model checker takes a new current state s and then calculate a new set *r_set* of starting points for the new checking cycle. If no state is available in the *Buffer*, the online model checker simply *resume*s the search work left in the last checking cycle.

Algorithm 4.2: Online Symbolic Model Checking Algorithm

```
input :  $M \times B_{\neg f} = (V, D, R^m, I, L, F^*), T, k$
output :  $safe, unsafe, unknown$
```

```
begin
  wait ($isEmpty(Buffer) == false$) //wait until $Buffer \neq \emptyset$
  set $timer$ to $T$ time units
  $s \longleftarrow Buffer$ //take the first monitored state
  $r\_set = \{r \in I \mid r.s = s\}$ //$s \in M$
  if $r\_set \wedge F^* \neq \emptyset$ then return $unsafe$
  $jump = 0$
  while $r\_set \neq \emptyset$ do
    $r'\_set = \{r' \in D \mid R^m(r, r') \wedge r \in r\_set\}$ //jump forward $m$ steps
    if $r'\_set \wedge F^* \neq \emptyset$ then return $unsafe$
    $jump{+}{+}$
    $r\_set = r'\_set$
    if $isTimeout(timer)$ then
      if $m * jump \geq k$ then output $safe$
      else output $unknown$
      set $timer$ to $T$ time units
      if $isEmpty(Buffer) == false$ then //$Buffer \neq \emptyset$
        $s \longleftarrow Buffer$ //take a new monitored state
        $r\_set = \{(s, q) \mid q \in Q \wedge s \models L_B(q)\}$ //$s \in M$
        $jump = 0$
      endif
    endif
  endwhile
end
```

BDD-based vs. SAT-based Search Engine

Both BDD (Binary Decision Diagram) [41] and CNF (Conjunctive Normal Form) can be used to represent symbolically the set of states and the transition relation between states. The online symbolic model checking algorithm can be implemented using BDD-based or/and SAT-based search engine.

The paper [42] compared the performance of the two search engines on 62 benchmarks drawn from commercial designs. The experimental results indicate that “SAT-based BMC is more effective than BDD-based BMC” for finding *shallow* counterexamples (of length, say, $k \leq 60$), while “BDD-based BMC is much faster” for finding *deep* counterexamples (of length, say, $k > 60$). In addition, the experiments done at IBM, Intel and Compag [22] confirm that “if k is small enough (typically not more than 60 to 80 cycles, depending on the model itself and the SAT solver), it outperforms BDD-based techniques.” “The deeper the bug is (i.e., the longer the shortest path leading to it is), the less advantage (SAT-based) BMC has.”

To our knowledge, it is more convenient to use BDD to calculate the (multi-step) successors of any set of states. But in some cases the size of the BDD representation (of some intermediate result) may be exponential independent of any variable ordering. By using SAT solver, the successors are not directly calculated, instead, the error paths of length k are searched (see Section 2.5). There are pros and cons to both BDD-based and SAT-based methods. In this thesis, we prefer SAT-based to BDD-based search engine.

4.2.3 Parallel Computing

By applying the symbolic model checking technique together with the multi-step transition relation R^m and the extended set F^* of the error states we are able to improve the performance of the online model checker to some degree. But this is still not the end of the story.

As mentioned in Chapter 1, we concern ourselves with the correctness of the embedded software applications. Many embedded applications are growing in complexity so as to fulfill more functionality. The underlying hardware demands higher performance but lower energy consumption. The interconnected multiple single core processors do not fit this need any more. As a result, "multicores have become an unavoidable reality"[66]. The added computational power is thus available for other purposes, say, online model checking, in our case. This can be fulfilled by installing additional (general-purpose) operating systems to gain more functionality.

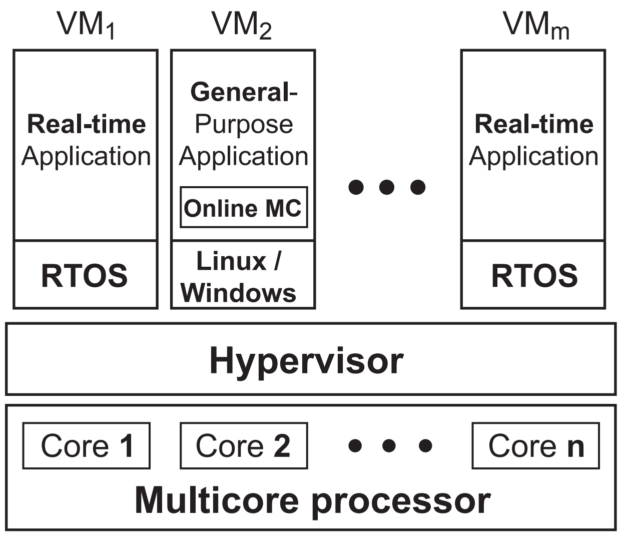

FIGURE 4.17: Multiple virtual machines (VMs) hosted on multicore processor

However, when the multiple operating systems are running on a multicore processor, it is a challenge to make them do not interfere with each other, e.g., shared memory access, interrupt handling, time management, and so on. Different virtualization techniques, such as container and virtualization [67], can solve this problem. Considering that many model checking tools work on top of general-purpose operating systems, a better solution is to insert an embedded virtualization layer below the multiple operating systems [68], as illustrated in Fig. 4.17. The hypervisor "virtualizes and arbitrates access to the cores, memory and devices" to ensure that "each operating system can properly execute in its own isolated partition" [68].

In this way, we are able to take the advantage of the parallelism of the multicore processor to improve the performance of the online model checker. Instead of just using one model checker, theoretically, two or more online model checkers can be used to run in parallel to speed up the online model checking process.

4.2.4 Prototype Implementation and Experimental Results

In cooperation with Mona Qanadilo and Sufyan Samara from An-Najah National University, Nablus, Palestine, we've implemented a simplified version of the online symbolic model checking algorithm using SAT solver [69]. The checking process is illustrated in Fig. 4.3. The starting points for each checking cycle (except the first one) are decided in a simple way similar to that of the explicit state-based implementation in Section 4.1.5.

We adopt a SAT-based instead of BDD-based search algorithm. On the one hand, the modern SAT solvers are now able to handle efficiently large SAT problems with hundreds of thousands of variables. According to [42], "SAT-based BMC is more effective than BDD-based BMC" for finding *shallow* counterexamples, in particular, "SAT solvers are quite effective in eliminating logic that is not relevant to a given property." This means that "SAT solvers appear to have significant potential for identifying that set of variables once a suitable property is given." On the other hand, the size of the BDD representation (of some intermediate result) may be exponential independent of any variable ordering in some special cases.

Optimizing SAT Solver for Online Model Checking

The SAT solver we used is zChaff[3], because we are more familiar with the implementation details of zChaff. First of all we need to tune zChaff for online model checking.

Converting a general propositional logic expression into its CNF representation usually needs to introduce many auxiliary variables and thus results in a larger formula with

[3]`http://www.princeton.edu/~chaff/zchaff.html`

excessive number of variables. The experimental results in [70] indicate that the original variables have more influence than the auxiliary variables on deduction. zChaff is a general SAT solver. It does not distinguish the original variables from the auxiliary ones in the given CNF representation. We'd like to make zChaff distinguish the original variables from the auxiliary ones and then give the original variables priority over the auxiliary variables in the assignment process of the SAT solver. For this purpose, we modified the decision strategy VSIDS (Variable State Independent Decaying Sum) of zChaff in favor of the original variables.

It is easy to see in Fig. 4.3 that the only difference among the individual SAT problems in different checking cycles is the initial condition $\mathcal{I}(\widehat{s_i})$, which indicates the values of the variables (of interest) monitored at runtime. Since each *conflict clause*[4] learned by the SAT solver is an implication of some clauses of the given SAT problem, it is redundant and has nothing to do with the valuation of the variables. Therefore, the conflict clauses learned in the previous checking cycles can be directly reused in the later checking cycles to reduce the space to be searched.

To switch from the current checking cycle to the next checking cycle, we'd like to make zChaff *restart* in an efficient manner. The restart operation in zChaff simply undoes the assignments of those variables at the decision levels greater than 0. Without loss of generality, let's assume that the monitored variables $v_1, v_2, \cdots, v_n$ be ordered in this way. The initial conditions $\mathcal{I}(\widehat{s_i})$ and $\mathcal{I}(\widehat{s_{i+1}})$ of any two checking cycles in a row usually have some common part, i.e., the valuation of some variables keeps unchanged. Therefore, we make zChaff backtrack to the first variable v_j (at the decision level j) whose valuation has been changed in the initial condition $\mathcal{I}(\widehat{s_{i+1}})$. Of course, if v_1 (at the decision level 1) is such a variable, then we have to backtrack to the decision level 1 in this case. However, as long as $j > 1$, we are able to reuse the deduction results done for $v_1, v_2, \cdots, v_{j-1}$ in the next checking cycle. In particular, if the initial conditions $\mathcal{I}(\widehat{s_i})$ and $\mathcal{I}(\widehat{s_{i+1}})$ happen to be the same, then we simply make zChaff resume the search work in the new checking cycle, provided that the work has not been completely done.

Experimental Results of Online Model Checking Using Parallel Computing

Theoretically, by introducing R^m and F^*, we are able to make the online symbolic model checker look ahead more steps in each checking cycle. A further improvement can be done using parallel computing. In doing so, we have multiple SAT-based online model checkers work in parallel on the 64 bit Windows platform with 2.13GHz i3 CPU and 4GB RAM.

[4]A redundant clause that captures the causes of an inconsistency discovered during the search (for a solution to the SAT problem) so as to prevent the same conflict from occurring again.

The case study is the MSI protocol with transient states taken from the NuSMV[5] software package. It is a basic cache-coherence protocol, which specifies that "There are three processors, each with one level of cache that stores 1-bit of data and has a 1-bit tag. The caches are write-back, write-allocate. The bus arbitration is round-robin. There is a memory with two 1-bit locations."

We generate the transition relation in CNF from the NuSMV specification of the MSI protocol and then unroll the transition relation up to k steps. Table. 4.2 lists the number of total variables, the number of original variables and the number of total clauses of $|[M]|_k$ for $k = 35, 40, 45$ and 50 respectively. The original variables account for about 24% of the total variables in each case. The property to be checked is an invariant of the form $\mathbf{AG}p$, from which the path constraint $|[C]|_k$ is derived in CNF. The monitored states are sampled every k steps from the execution trace generated simply by "executing" the model itself. Thus, we have $s_i = \widehat{s_i}$.

k	Total Variables	Original Variables	Total Clauses
35	6662	1620	22663
40	7602	1845	25873
45	8542	2070	29083
50	9482	2295	32293

TABLE 4.2: MSI model $|[M]|_k$ with path of different lengths

The decision strategy adopted in the SAT solver has a large effect on the performance of the online model checker. Should we use different decision strategies or use the same decision strategy in the SAT-based online model checkers running in parallel? To answer this question, we make two SAT solvers use different decision strategies: the original VSIDS and the modified VSIDS (in favor of the original variables). Then, we do online model checking for the MSI protocol with $k = 35, 40, 45$ and 50 from 201 different initial states, i.e., 201 checking cycles. In this experiment, we do not set any time limit for each checking cycle, just let the online model checkers run to the end and then start the new checking cycle. From one checking cycle to the next checking cycle, the learned clauses are reused by the SAT solvers. If one SAT solver works faster than the other one, it will "consume" more initial states than the other SAT solver, i.e., it will run more checking rounds. This is confirmed by the experimental results illustrated in Fig. 4.18 and in Fig. 4.19. The former shows that the SAT solver with modified VSIDS decision strategy runs more checking cycles in all the four cases; the latter shows the cumulative execution time (in seconds) of each SAT solver in all the four cases. It is easy to see that the SAT solver with modified VSIDS decision strategy takes less execution time while it runs more checking rounds in three of the four cases.

[5]http://nusmv.fbk.eu

In this experiment, we use Round Robin scheduling algorithm, which gives each online model checker equal priority. Although the two online model checkers do not run on a real parallel hardware system, the experimental results do reflect the fact that the decision strategy in favor of the original variables has a better performance on average, which conforms with the conclusion in [70]. Consequently, we adopt the same decision strategy (i.e., modified VSIDS) for the SAT solvers working in parallel.

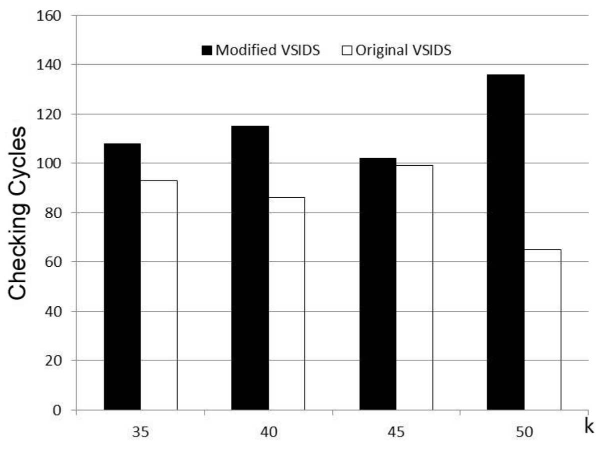

FIGURE 4.18: Performance of the two decision strategies

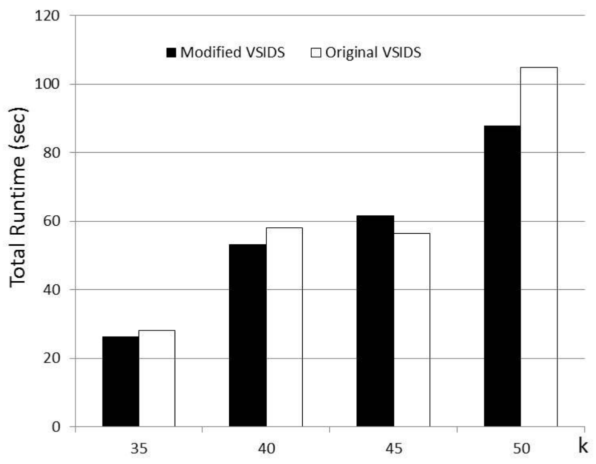

FIGURE 4.19: Cumulative runtime of the two decision strategies

In addition, Fig. 4.20 shows the performance comparison of the modified VSIDS strategy with the original VSIDS and the static order strategy obtained by online checking the MSI protocol for $k = 50$ from 201 different (monitored) states. We zoom in on the results obtained in the checking cycles from 90 to 128. It is easy to see that the performance of the modified VSIDS strategy on average is more stable than the other two strategies in the sense that there exist no large fluctuations in execution time from one checking cycle to the next.

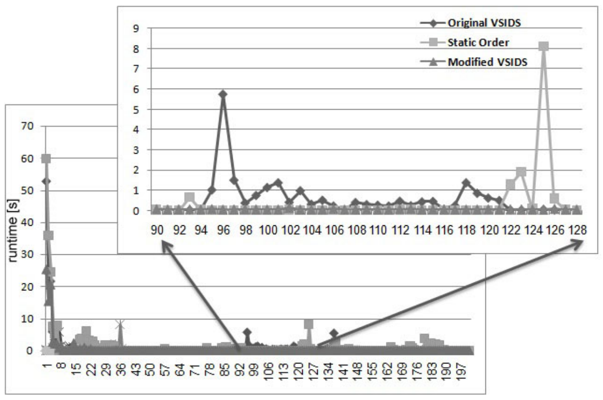

FIGURE 4.20: Performance of the three decision strategies for $k = 50$

How many online model checkers running in parallel can obtain a better performance? We use 1, 2, 3, 4 and 5 SAT-based model checkers respectively to online check the MSI protocol for $k = 50$ from 201 different initial states (i.e., 201 checking cycles). In this experiment, we also do not set any time limit for each checking cycle, just let the SAT solvers run to the end and then start the new checking cycle. Given a setting of multiple model checkers working in parallel, the maximum cumulative runtime of them indicates the performance of this setting. E.g., in the setting of 5 SAT-based model checkers, we take the maximum cumulative runtime of the 5 SAT solvers. The experimental results in Fig. 4.21 illustrate the maximum cumulative runtime (in seconds) of the SAT solver(s) in each setting.

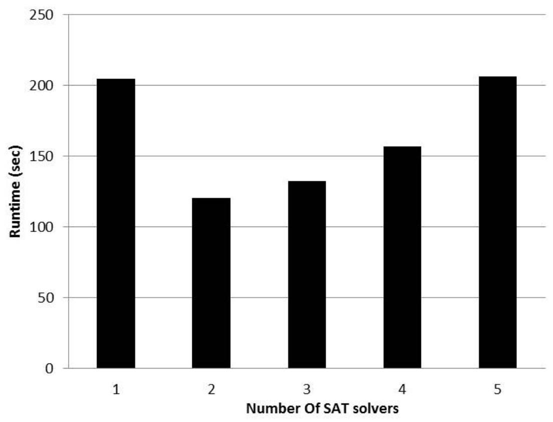

FIGURE 4.21: Performance comparison of multiple SAT-based model checkers

It is easy to observe that the setting of 2 SAT-based model checkers outperforms all the other four settings. The possible reasons are analyzed as follows:

- The experimental results indicate that the learned clauses shared between different checking cycles do improve the performance of the SAT solvers. But the more SAT solvers working in parallel, the less (initial) states each SAT solver can consume, thus the less the learned clauses are produced to prune the state space for each SAT solver.

- Each SAT solver has to access cache and memory frequently. We use Round Robin scheduling algorithm, which gives each model checker equal priority. The more SAT solvers working in parallel, the higher the probability of access conflicts due to requesting data at the same time. The memory is a shared resource among different SAT solvers. If we run more than two SAT-based model checkers in parallel, it means more synchronization overhead. In this sense, our platform does not simulate the parallel feature very well.

In this experiment, we notice that the learned clauses have a large impact on the performance of the SAT solver. In the setting of 5 SAT-based model checkers, one SAT solver consumes only 8 (initial) states, but its total execution time is even more than that of the single solver setting, in which 201 (initial) states are processed.

Considering the price-performance ratio, it is better to use 2 SAT-based model checkers working in parallel. In addition, we've tried to make the 2 SAT solvers share the shortest learned clauses with each other, but the performance improvement is not satisfying due to the additional communication overhead. Therefore, we keep the 2 SAT-based model checkers working independent of each other.

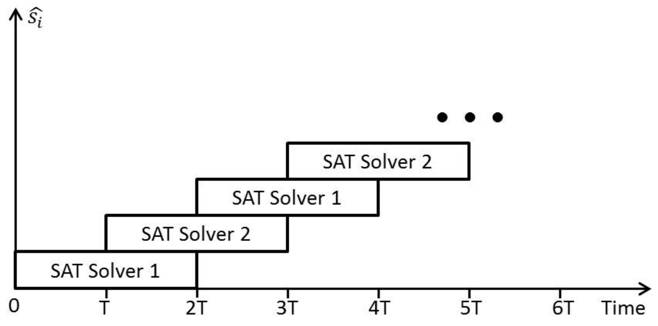

FIGURE 4.22: Two online model checkers working in parallel

Now let two SAT-based model checkers work in parallel as illustrated in Fig. 4.22. Every T time units only one model checker is allowed to take a state from the ring buffer. Each model checker has $2T$ time units to do the search work. To simulate a general buffer setting, we use a randomly generated *Boolean* value to decide whether the ring buffer is *empty* or not. For each model checker, if the ring buffer is *empty*, the model checker will either resume the search work of the last checking cycle, provided that the work has not finished yet; or else simply wait until the next checking cycle starts. Otherwise, there is a state, say $\widehat{s_i}$, available, then within the given $2T$ time units, the online model checker tries to search for an error path of length up to k steps in the partial state space derived from $\widehat{s_i}$.

Given the model derived from the MSI protocol for $k = 35$, the experimental results in Fig. 4.23 demonstrate the execution of one online model checker in the checking cycles from 33 to 65 with the predefined time limit $2T = 0.08s$. The highlights are described as follows:

- In the checking cycles 45, 47, 49 and 53 the online model checker can not get a state from the ring buffer and the search work of the last checking cycle has been done, therefore, it does nothing but waits for the next checking cycle.
- In the checking cycles 37 up to 40 the online model checker can not get a state from the ring buffer, since the search work of the checking cycle 36 has not finished yet, therefore, it resumes the search work of the checking cycle 36.
- In the checking cycle 44 the online model checker resumes the search work of the checking cycle 43 and finally gets a definite result this time, i.e., no error path of length $k \leq 35$ starting from the given state is found.

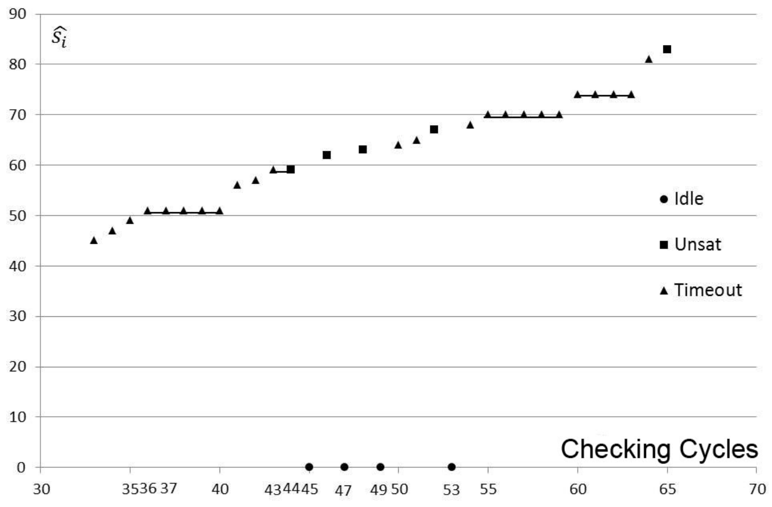

FIGURE 4.23: Execution of one online model checker with the random buffer setting

Let T be the predefined time limit for each checking round. If the online model checker does not finish the search work within this limited time, it is usually difficult to measure what percentage of the state space of the behavioral model has been searched within the given T time units. There is a simple way to estimate this percentage coarsely. Given a monitored state s_i, we have the SAT solver run to the end to get the total runtime in the bounded model derived from s_i, denoted as T_{total}. Then, T/T_{total} estimates coarsely the percentage of the state space derived from s_i that has been searched by the model checker in one checking cycle.

Let's take the MSI protocol for $k = 20$ as example, which has 3842 variables (including 945 original variables) and 13033 clauses. Recall that for online model checking we do *not* need to set k to a large number (thanks to the offline forward and/or backward exploration). For an initial state, the total runtime T_{total} we measured is about $1.6s$. Then, $T = 0.08s$ indicates that about $0.08/1.6 = 5\%$ of the state space of the bounded model derived from this state has been searched in one checking cycle.

4.3 Summary

In this chapter we present the concept of online model checking, which is the main topic of this thesis. Online model checking is a lightweight and incomplete method applied at runtime to falsify, instead of verify, the given behavioral model against the LTL property to be checked. The errors found in the behavioral model may indicate the errors in the source code of the target software system.

In theory both safety and liveness properties can be checked during system execution by online model checking. The basic idea is to reduce the online model checking problem to the invariant checking problem, which can be solved by online reachability analysis. Doing model checking online suffers from the limited execution time allocated to each checking cycle. Its performance depends mainly on the search algorithm and the underlying hardware as well as the complexity of the problem to be checked. The workload of the online model checker can be reduced to some degree by introducing the m-step transition relation R^m and the n-step backward reachable set F^*. The performance of the online model checker can be further improved by using a symbolic state-based search engine and making more than one online model checker working in parallel.

As a proof of concept, we've implemented an explicit state-based and a symbolic state-based online model checking algorithm. The explicit state-based online model checker is mainly used to observe the internal structure of the behavioral model and its influence on the performance of the online model checker. The experimental results indicate that the density of states in the model and the number of state variables are the two main factors affecting the speed of the online model checker. The symbolic state-based online model checker adopts a tailored SAT-solver as its search engine. The experimental results in different parallel settings indicate that two online model checkers working in parallel can offer a better price/performance ratio.

Chapter 5

Model Generation and Source Code Instrumentation

We'd like to apply the online model checking mechanism to embedded control software. In practice, a software application after release may still contain subtle errors that have escaped from being detected in the development life cycle. For a safety-critical system, it is meaningful to catch in time these deep, corner-case errors in the embedded software during system execution. As explained in Chapter 4, given the source code of the program to be checked, in order to do online model checking, we need to instrument the source code with finitely many monitoring points (once and for all) on the one hand, and to obtain a (behavioral) model of the target program on the other hand. In this chapter, we focus on model generation and source code instrumentation.

5.1 Embedded Control Applications

Embedded control software is used to control an external environment, which may be artificial or biological objects, e.g., physical plants and humans. This environment (i.e., the controlled objects) is connected to the computer system, on which the control program is running, through sensors and actuators or through other input-output interfaces. The execution of such a kind of software application must meet various timing and other constraints that are imposed on it by the (real-time) behaviors of the controlled objects. But this doesn't mean that the control program has to run as fast as possible. Instead, it means that all the tasks of the control program should finish execution in the worst-case by their deadlines. The deadlines are dictated by the controlled objects. A deadline is *hard*, if missing it may result in severe consequence(s); otherwise, it is *soft* [2].

Our concern is the software applications in such domains as automotive, aerospace, manufacturing control, and the like, which exhibit time-periodic and multimodal features. Modern control software in these domains is becoming increasingly complicated due to the following requirements [71] on it:

- operate in distributed and embedded computing environments;
- communicate through different protocols;
- adapt to changes in operating environments; and
- behave in a dependable manner for safety critical systems.

A platform-independent programming abstraction is presented in [72] for specifying the behaviors of embedded control software with (hard) real-time constraints. According to [72], a typical control application consists usually of periodic (software) tasks governed by a mode-switching logic for activating or deactivating tasks. Each (operational) mode contains a (fixed) set of tasks active in this mode.

E.g., a typical fly-by-wire control system has five operational modes: takeoff, cruise, autopilot, degraded and landing modes. In each of these modes, different sensing tasks, control laws, and actuating tasks need to be executed. That is, some tasks are added to, while others are removed from, the task set of the current mode. For instance, in the takeoff mode, the landing gear is not needed; in the autopilot mode, the inputs are taken from a supervisory flight planner, instead of from the pilot's stick; and in the degraded mode, some of the sensors and actuators are disabled due to damage.

Intuitively, modes impose some structure on the behaviors of the given control application [73]. The control application is in one mode at a time, i.e., there is only one active mode at any given time. The mode-switching logic determines the possible transitions from one mode to other modes. It is a kind of finite state machine, in which each state represents a mode and each transition specifies a possible switching between two different modes. A mode switch can result in removing some tasks and adding others. After a mode switch is finished, all subsequent actions follow the definition of the target mode. A mode switch may happen while a task is still running. In this case, this task needs to be continued in the target mode.

In each mode, a set of tasks are executed periodically following some scheduling strategy. A task is a piece of application-level code, which typically implements a control algorithm. Tasks communicate with each other as well as with sensors and actuators by drivers. A driver is a piece of system-level code, which provides an abstract interface to (physical) devices by transporting and converting values between ports. A port is abstracted as a typed variable. It keeps its value over time, until it is updated.

There are sensor ports, actuator ports, and task ports as well as mode ports. The sensor ports are updated by the environment, i.e. sensor ports are treated as elementary sources of values. All other ports are updated by their respective drivers. A driver may provide sensor readings for the input ports of a task; or load task results into actuator ports; or provide task results for the input ports of other tasks.

The task ports are used to communicate data between concurrent tasks or transfer data from one mode to the next. In general, a task has input ports and output ports. A typical task has also an internal state, the values at which are inaccessible outside the task. What's more, a task has a function from its input ports and the current state to its output ports and the next state. This function is usually implemented by a sequential program written in some programming language, say, C code.

In effect, a driver is a function that converts values of sensor ports and mode ports of the current mode to the values for the input ports of the tasks in the target mode, or loads the input ports with constants. Once the input ports of a task are updated by a driver, the task is ready to run. That is, the task is put into a ready queue, from which the scheduler of the operating system chooses tasks for execution following some scheduling scheme.

Generally, a task may be mapped to a software-programmable component (in most cases) with a local operating system, e.g., general-purpose processor (CPP) or application-specific instruction set processor (ASIP). A task may also be mapped to a hardware component, e.g., field-programmable gate array (FPGA) or application-specific integrated circuit (ASIC).

In this thesis, we are concerned with the tasks running on top of some real-time operating system.

5.2 Model Generation

A model is an abstraction of some aspect of a system [71]. The (bounded) models for online model checking can be obtained at system development phase or by abstraction from the source code of the program under investigation.

To deal with the increasing complexity of software products, Model-based Development (MBD) becomes more and more popular in developing software applications in the automotive, aerospace and other industries. The investigation [74] shows that using MBD can help to shorten the development *time*, reduce the development *costs*, improve the product *quality*, just to mention some aspects. MBD aims to reduce the complexity by means of building the design models at multiple levels of abstraction and/or from different perspectives, analyzing the models, and transforming the models into concrete (code) implementation [71]. Thus, for a software application developed using MBD, the design models at different levels of abstraction can be reused for online model checking.

It's worth pointing out that using MBD does allow verification activities such as model checking, theorem proving, simulation and testing to be conducted at the model level in the early design phase. Many errors can thus be detected and corrected early in the development life cycle, and this, in turn, does decrease the number of errors discovered during the integration or system test phase. However, for large complex systems, it is still difficult to prove the correctness of the behavioral model(s) due to the state space explosion problem. E.g., for composing two mode transition diagrams of a flight guidance system even in a constrained way, "there are typically over 10^{20} reachable states" [75].

On the other hand, the code generator used by MBD is usually too complex to be proved for correctness. According to [74], "some companies are the opinion that the current code generators are not applicable to generate high safety relevant code." Indeed, not all software applications are developed using MBD. Traditional hand-coded software development is still useful, especially for developing ultrahigh safety critical systems. In this case, the behavioral models at different levels of abstraction can be generated directly from the source code of the software application to be checked. In fact, source code itself is a kind of model describing how the program will behave when executed [71].

As mentioned in Section 5.1, a mode transition diagram is a kind of finite state machine, which is generated during the software development phase. Online model checking can also be used to verify the mode-switching logic against the desired properties specified in LTL. The execution result of a task in the current mode may trigger a transition to some next mode. In this sense, each task can be seen as atomic. A typical task is implemented by a sequential program written, say, in C code, which repeats the following three steps: receive input data, update internal state and produce output. Therefore, the monitoring point can be inserted "between the portions of C code which produce output and receive new input" [76].

In this thesis the attention is paid to checking the functional correctness of a task, which is identified as important among the others in the task set, during program execution. Since the focus is on the internal behaviors of the task to be checked, the timeliness of the task is beyond our concern. It is up to the scheduler to ensure the timeliness of the tasks in the system.

5.3 Source Code Instrumentation

We need to identify a finite set of monitoring points in the source code of the program to be checked. For the sake of simplicity, we suppose that the target program is written in a restricted subset of C that conforms to the (mandatory) requirements of the MISRA-C coding standard. The monitoring points are determined by partitioning the control flow graph derived from the source code into a finite set of subgraphs.

5.3.1 MISRA C

The C programming language is widely used to develop the embedded software for safety critical systems in industry. However, some linguistic features of the C language are either specified indefinitely or implementation-dependent. To avoid the traps and pitfalls of C, guidelines for writing safe, portable and reliable code in C are highly demanded. The Motor Industry Software Reliability Association[1] (MISRA) published in 1998 a coding standard for the C language officially known as MISRA-C:1998 [77]. MISRA-C:2004 titled "Guidelines for the use of the C language in critical systems" [78] was released and in 2013 the release of MISRA C:2012 [79] was announced. Nowadays MISAR C has evolved into a de-facto coding standard for developing embedded software not only in the automotive industry, but also in other industries such as aerospace, railways, nuclear, defense and medical devices, to name just a few.

To promote the safest possible use of C, the MISRA C standard recommends a restricted subset of C, which has already established practices in industry. MISRA-C:2004 contains 122 *mandatory* and 20 *advisory* rules. The most significant limitations of MISRA-C:2004 [78] are listed below:

Rule 16.2 (required):	Functions shall not call themselves, either directly or indirectly.
Rule 20.4 (required):	Dynamic heap memory allocation shall not be used.

These two rules simplify the structure of the states of the target program. Generally, a program state is identified by the following two parts [28]:

- a control component — a point of control PC, which is not simply the program counter, but may contain a procedure-calling chain of the target program; and
- a data component — an assignment of values to all the variables, including input data and internal data, at the given point of control of the target program.

[1]`http://www.misra-c.com`

The points of control are located before (or after) the smallest execution units of the source code of the target program. The data types declared in the program determine the smallest distinguishable data components. The Cartesian product of the definition ranges of all the state components form the state space of the target program. Notice that this definition covers the states that may never be reached by the program's execution for any set of input data. It is indeed undecidable whether a given state may be reached or not [28].

A program written in a language that allows dynamic memory allocation and (procedure) recursion results in dynamic data structure and dynamic PC structure respectively for identifying the states of the program. This increases the complexity of the monitoring operations. In addition, the resulting state space is usually infinite.

On the contrary, a program written in a language that allows only static memory allocation and no recursion produces a finite state space[2] with simple state structure, which is relatively easy to monitor, because the correspondence of the program variables to the memory addresses can be established once and for all at compile-time.

To simplify the problem, we assume that the embedded software under consideration is written in C and compliant to the MISRA C coding standard. Although in specific cases some rules of MISRA C may be deviated from, in this thesis we suppose that the above mentioned two rules are always obeyed.

5.3.2 Control Flow Graph

Among the points of control of the target program, we need to select some of them as monitoring points for online model checking. This is done by analyzing the control flow graph of the target program.

Given a sequential C program P with function calls (if any) managed by means of macro-expansion. Since the statements of a sequential program are ordered one after another, there is a unique *entry* point and a unique *exit* point for each statement, i.e., the smallest execution unit of the source code. To avoid redundancy, the exit point of a statement and the entry point of the following statement are merged into one. Together with the entry and the exit points of the program P itself, we are able to uniquely label the entry and the exit points of all statements of P [18, 80].

Without loss of generality, let's represent the C program as $\mathsf{P} = (\mathsf{V}, \mathsf{L}, l_0, \mathsf{T})$ [81], where V is the set of typed variables that constitute the data component of the program state, L is the set of locations, i.e., points of control in the target program, l_0 is the entry

[2]Provided that all the data types are defined with finite ranges in the program.

point of the program P, and T is the set of transitions between the control points. Each transition is defined as a tuple (l, c, l') with $l, l' \in \mathsf{L}$ and c a constraint over the free variables in $\mathsf{V} \cup \mathsf{V}'$. The variables in V record the values at the (present) control location l, while the variables in V' record the values (of the variables from V) at the (next) control location l'. Obviously, the set L of locations and the set T of transitions together decide a directed (cyclic) graph (see Fig. 5.4 (a)), called the control flow graph[3] (CFG) of the program P, denoted as $\mathsf{G}(\mathsf{P}) = (\mathsf{L}, \mathsf{T})$.

In the terminology of (control flow) graph [82], we also call L the set of vertices and T the set of edges of the graph $\mathsf{G}(\mathsf{P})$. $\mathsf{G}(\mathsf{P})$ contains a finite set of vertices and a finite set of edges in it. A vertex l_i is said to *dominate* (or to be a *dominator* of) a vertex l_j if every path from the *entry* vertex l_0 to l_j must go through l_i. By definition, every vertex dominates itself and the entry vertex l_0 must be a dominator. A vertex l_i *strictly dominates* a vertex l_j if l_i dominates l_j and $l_i \neq l_j$. A *immediate dominator* of a vertex l_j is a vertex l_i that strictly dominates l_j but does not strictly dominate any other vertex between l_i and l_j. The immediate dominator of a vertex (other than l_0) is unique. E.g., in Fig. 5.4 (a) the vertices **1**, **2**, and **4** (strictly) dominates the vertices **5** and **6**, while the vertex **4** is an immediate dominator of the vertices **5** and **6**.

Given a vertex $l \in \mathsf{L}$, and an integer $k > 0$, we define $\mathsf{G}(l, k)$ as a k-bounded tree obtained by unwinding the graph $\mathsf{G}(\mathsf{P})$ starting from l up to k steps (see Fig. 5.4 (b), (c), and (d) for example). We call the vertices of this tree as *nodes* to distinguish them from the vertices of the graph, from which the tree is derived. The vertex l is the root node of the tree $\mathsf{G}(l, k)$. If a vertex in $\mathsf{G}(\mathsf{P})$ is reached along different paths within k steps starting from l, there may exist several nodes in the tree $\mathsf{G}(l, k)$ representing the same vertex in $\mathsf{G}(\mathsf{P})$. E.g., the node **4** occurs twice in the tree $\mathsf{G}(1, 3)$ in Fig. 5.4 (b). The *maximum depth* of a node $l_i \in \mathsf{L}$ with respect to the root node l, denoted as $\delta_{max}(l, l_i)$, is the number of edges on the longest path from l to l_i in the tree. Obviously, $\delta_{max}(l, l_i) \in [0, k]$. E.g., in $\mathsf{G}(1, 3)$ (see Fig. 5.4 (b)) $\delta_{max}(1, 4) = 3$.

5.3.3 Graph Partitioning

As mentioned in subsection 4.1.4, we'd better select such a set of vertices in $\mathsf{G}(\mathsf{P})$ as monitoring points that satisfy the following conditions: (i) any two adjacent monitoring points are at distance at most k steps; (ii) any vertex other than monitoring point in $\mathsf{G}(\mathsf{P})$ is at distance at most k steps from some monitoring point; and (iii) between any two vertices with distance greater than k steps there must exist some monitoring point.

[3]Note that this definition is different from the traditional definition of control flow graph, where each node represents a basic block (of code) and each (directed) edge a jump in the control flow.

It is easy to reason that there always exists a solution, e.g., a trivial solution is to set all the vertices in $\mathsf{G(P)}$ as monitoring points. Of course, the solution is not unique. Given an integer $k > 0$, besides the entry point l_0, we'd like to determine a smaller set of vertices $\{l_1, l_2, \cdots, l_n\} \subset \mathsf{L}$ such that the trees $\mathsf{G}_0(l_0, k), \mathsf{G}_1(l_1, k), \mathsf{G}_2(l_2, k), \cdots, \mathsf{G}_n(l_n, k)$ can cover the graph $\mathsf{G(P)}$ and have a similar size in terms of nodes and edges. In other words, we hope to find out a smaller set of monitoring points that can be distributed in $\mathsf{G(P)}$ as evenly as possible.

For this purpose, we'd like to partition $\mathsf{G(P)}$ into finite subgraphs in similar size (more or less). There are two common approaches [83] to partitioning a graph: *edge-cut* and *vertex-cut*. The former determines a set of edges (called edge-cut), while the latter determines a set of vertices (called vertex-cut), whose deletion can make the graph disconnected. Fig. 5.1 illustrates the difference of the two partitioning methods. For each edge (x, y) in an edge-cut (see Fig. 5.1(a)), the two vertices x and y serves as the *exit*- and *entry*-points of the respective partitions. For each vertex x in a vertex-cut (see Fig. 5.1(b)), the vertex x serves as both the *exit*- and *entry*-points of the respective partitions. In addition, a vertex can be cut in multiple ways as shown in Fig. 5.2, while an edge can only be cut in one way.

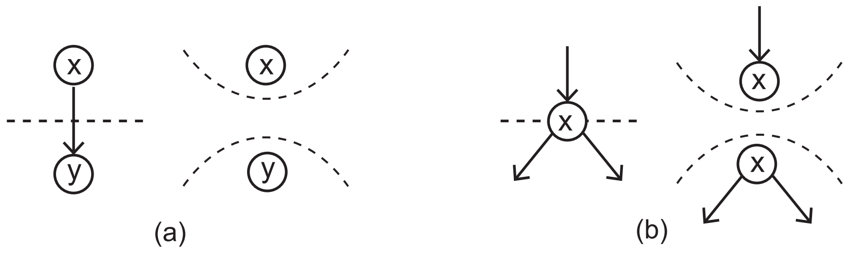

FIGURE 5.1: (a) Edge-cut partitioning vs. (b) Vertex-cut partitioning [83]

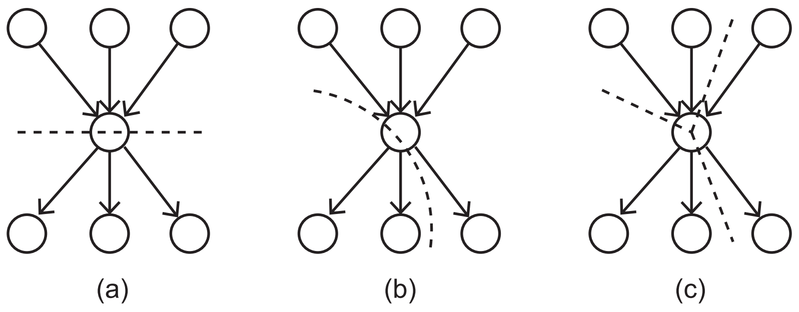

FIGURE 5.2: Cutting a vertex in different ways [83]

Edge-cut partitioning is not suitable for us, because the edges in the cut does not belong to any partition, e.g., the edge from x to y in Fig. 5.1(a), thus they can not be reached by the online model checker without specifically dealing with them. Vertex-cut partitioning does not have this problem. The vertices in the cut belong to at least one partition (see Fig. 5.1(b)). All the partitions together cover the whole graph with neither vertices nor edges left outside. Therefore, we partition $\mathsf{G}(\mathsf{P}) = (\mathsf{L}, \mathsf{T})$ by vertex-cut partitioning.

It is usually NP-hard to find out a minimum vertex-cut that leads to partitions of similar size. Here we give a simple procedure to produce a smaller vertex-cut for partitioning $\mathsf{G}(\mathsf{P})$ into *k-bounded* trees. The vertices in the cut are then selected as monitoring points.

It is observed that for any vertex x in the vertex-cut, the tree $\mathsf{G}(x, k)$ must cover some other vertices in the vertex-cut, and these vertices form a local cut of $\mathsf{G}(x, k)$. E.g., for $k = 3$, a possible vertex-cut of the (control flow) graph in Fig. 5.4 (a) is $\{\mathbf{1}, \mathbf{4}, \mathbf{7}\}$, then the tree $\mathsf{G}(1, 3)$ covers the vertex **4** which forms a local cut in $\mathsf{G}(1, 3)$ (see Fig. 5.4 (b)), and the same goes to the trees $\mathsf{G}(4, 3)$ and $\mathsf{G}(7, 3)$, where the vertex **7** forms a local cut in $\mathsf{G}(4, 3)$ (see Fig. 5.4 (c)) and the vertex **4** a local cut in $\mathsf{G}(7, 3)$ (see Fig. 5.4 (d)).

Therefore, we aim to identify a smaller local cut (denoted by the dashed line) closer to the *bottom line* of $\mathsf{G}(x, k)$ as illustrated in Fig. 5.3. That is, given a non-negative integer ε (relatively far) less than k, we say that a local cut of $\mathsf{G}(x, k)$ is *close* to the bottom line of $\mathsf{G}(x, k)$ with respect to ε, if for any node y in the cut, we have $k - \delta_{max}(x, y) \leq \varepsilon$.

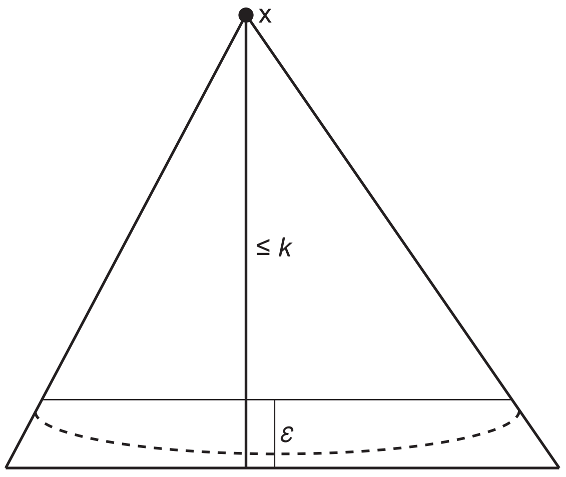

FIGURE 5.3: A smaller cut close to the bottom line of $\mathsf{G}(x, k)$

We say that a node y in the tree $\mathsf{G}(x, k)$ is *leaf node*, if y is either an exit point of the program P, or else $\delta_{max}(x, y) = k$. In the former case, $\delta_{max}(x, y) \leq k$. E.g., in the tree $\mathsf{G}(7, 3)$ (see Fig. 5.4 (d)) the leaf nodes are **5**, **6**, and **9**, because $\delta_{max}(7, 5) = 3$ and $\delta_{max}(7, 6) = 3$, but $\delta_{max}(7, 9) = 2$ which is less than $k = 3$, this is allowed, because the node **9** is an exit point of the program.

The bottom line of $\mathsf{G}(x, k)$ consists of those non-trivial leaf nodes of $\mathsf{G}(x, k)$ that are *not* exit points of the program P. Obviously, if all the leaf nodes of $\mathsf{G}(x, k)$ are exit points of the program P, then there is not need to cut $\mathsf{G}(x,k)$, because the tree $\mathsf{G}(x,k)$ can not "grow" further, thus the bottom line is meaningless in this case.

The bottom line of $\mathsf{G}(x,k)$ itself can be seen as a local cut at the depth k relative to the root node x. We'd like to evolve it into a smaller cut of $\mathsf{G}(x,k)$ by applying repeatedly to it the following two rules:

rule 1: for a node y in the cut, if y dominates other node(s) in the cut, then remove those nodes dominated by y from the cut;

rule 2: for a node y *not* in the cut, if y *immediate* dominates more than one node in the cut and $k - \delta_{max}(x,y) \leq \varepsilon$, then remove those nodes *immediate* dominated by y from the cut and add y to the cut.

Let *global_cut* be a set of nodes forming a vertex-cut of $\mathsf{G}(\mathsf{P})$ and *local_cut* a set nodes forming a vertex-cut of $\mathsf{G}(x,k)$. We partition $\mathsf{G}(\mathsf{P})$ in a way described in Algorithm 5.1. First of all, *global_cut* is initialized to $\{l_0\}$, the entry point of the program P. For each node $x \in$ *global_cut*, which has not been processed yet, the bottom line of $\mathsf{G}(x,k)$ with respect to *global_cut* is calculated in Algorithm 5.2. If the local cut of $\mathsf{G}(x,k)$ is *not* empty, we try to reduced it to a smaller cut close to the bottom line of $\mathsf{G}(x,k)$ with respect to ε. Afterwards, the smaller *local_cut* is added to *global_cut*. This procedure is repeated until all the nodes in *global_cut* have been processed. At this time, the nodes in *global_cut* are the vertex-cut of $\mathsf{G}(\mathsf{P})$.

Algorithm 5.1: Partition control flow graph $\mathsf{G}(\mathsf{P})$

```
input : $\mathsf{G}(\mathsf{P}), k, \varepsilon$
output : global_cut
begin
  global_cut = $\{l_0\}$ //initialize global vertex-cut
  while (there exists unprocessed node in global_cut) do
    let $x$ be an unprocessed node in global_cut
    local_cut = bottom_line($\mathsf{G}(x,k)$) //see Algorithm 5.2
    if (local_cut $\neq \emptyset$) then //reduce the size of local_cut
      repeatedly apply rule 1 and rule 2 until fixed-point
      add local_cut to global_cut
    endif
  endwhile
end
```

Let *exit_points* be the set of exit points of the program P. Given *global_cut*, i.e., a set of thus far identified nodes in the vertex-cut of $\mathsf{G}(\mathsf{P})$, we say that a node z in $\mathsf{G}(x,k)$ is a *leaf* node with respect to *global_cut*, if $z \in$ *exit_points*, or $z \in$ *global_cut*, or else

$\delta_{max}(x, z) = k$. That is, taking *global_cut* into account, we redefine the bottom line of $\mathsf{G}(x, k)$ as those non-trivial leaf nodes of $\mathsf{G}(x, k)$ that belongs neither to *exit_points* nor to *global_cut*. Algorithm 5.2 returns such kind of nodes as *local_cut* of $\mathsf{G}(x, k)$.

ALGORITHM 5.2: Calculate the bottom line of $\mathsf{G}(x, k)$ with respect to *global_cut*

```
input : G(x,k), global_cut, exit_points
output : local_cut
begin
  current_set = {x} //initialize local vertex-cut
  next_set = ∅
  for (step = 1 to k) do
    while (there exists unprocessed node in current_set) do
      let y be an unprocessed node in current_set
      for (each immediate successor z of y) do
        if (z ∉ global_cut) and (z ∉ exit_points) then
          add z to next_set
        endif
      endfor
      if (next_set == ∅) then return ∅
    endwhile
    current_set = next_set
    next_set = ∅
  endfor
  return current_set
end
```

The **for** loop in Algorithm 5.2 iterates no more than k times. In each iteration, the successor nodes of the nodes in *current_set* are calculated, whose number is not more than that of the nodes in the tree $\mathsf{G}(x, k)$. Therefore, the time complexity of Algorithm 5.2 is linear in the number of nodes in the tree $\mathsf{G}(x, k)$. This algorithm returns either a *empty* set or a set of nodes neither in *global_cut* nor in *exit_points*. In the latter case, the algorithm ensures that for any leaf node y it returns, there exists at least one path from the root node x to the leaf node y such that no nodes between them are in *global_cut* or *exit_points*.

For each tree $\mathsf{G}(x, k)$ with a (non-empty) set of nodes returned by Algorithm 5.2, Algorithm 5.1 tries to reduce the size of its *local_cut* by applying repeatedly **rule** 1 and **rule** 2 as long as possible. The size of the final *local_cut* is not more than the number of nodes on the bottom line of $\mathsf{G}(x, k)$. The nodes in the *local_cut* are finally added to *global_cut*. As a consequence, the size of *global_cut* increases monotonically. Since the number of nodes in the graph $\mathsf{G}(\mathsf{P})$ is finite, Algorithm 5.1 can definitely terminate. On the other hand, Algorithm 5.1 ensures that all the trees produced can cover the whole graph $\mathsf{G(P)}$. Therefore, the time complexity of this algorithm is linear in the number of the vertices in $\mathsf{G(P)}$.

Let's use the control flow graph $\mathsf{G(P)}$ illustrated in Fig. 5.4 (a) as an example to explain our partitioning algorithm. The vertices of $\mathsf{G(P)}$ are named by numbering. Given $k = 3$ and $\varepsilon = 1$, we get the tree $\mathsf{G}(1, 3)$ shown in Fig. 5.4 (b) by unfolding 3 steps starting from the vertex **1**. Notice that the vertex **4** is duplicated twice in $\mathsf{G}(1, 3)$. According to Algorithm 5.2, the bottom line of $\mathsf{G}(1, 3)$ consists of $\{\mathbf{4}, \mathbf{5}, \mathbf{6}\}$. In Algorithm 5.1, *local_cut* is initially set to the bottom line of $\mathsf{G}(1, 3)$. Since the node **4** dominates the other two nodes **5** and **6**, it is safe to remove these two nodes from *local_cut* (**rule 1**). Now the local cut of $\mathsf{G}(1, 3)$ is reduced to $\{\mathbf{4}\}$. No further reduction is possible, therefore, we add it to the global vertex-cut, i.e., *global_cut* $= \{\mathbf{1}, \mathbf{4}\}$. Similarly, we get the tree $\mathsf{G}(4, 3)$ shown in Fig. 5.4 (c). The bottom line of $\mathsf{G}(4, 3)$ is $\{\mathbf{3}, \mathbf{8}\}$. Since the node **7** immediate dominates these two nodes, it is safe to delete them and then add the node **7** to the local cut of $\mathsf{G}(4, 3)$ (**rule 2**). As a result, we have *global_cut* $= \{\mathbf{1}, \mathbf{4}, \mathbf{7}\}$. Finally, we get the tree $\mathsf{G}(7, 3)$ shown in Fig. 5.4 (d). The bottom line of $\mathsf{G}(7, 3)$ is *empty* (with respect to *global_cut*) because the node $\mathbf{4} \in$ *global_cut* and the node $\mathbf{9} \in$ *exit_points* (see the lines 7-11 in Algorithm 5.2). By now all the nodes in *global_cut* have been processed, therefore, $\mathsf{G(P)}$ is partitioned into three components $\mathsf{G}(1, 3)$, $\mathsf{G}(4, 3)$ and $\mathsf{G}(7, 3)$. That is, the monitoring points in the target program are located in $\{\mathbf{1}, \mathbf{4}, \mathbf{7}\}$.

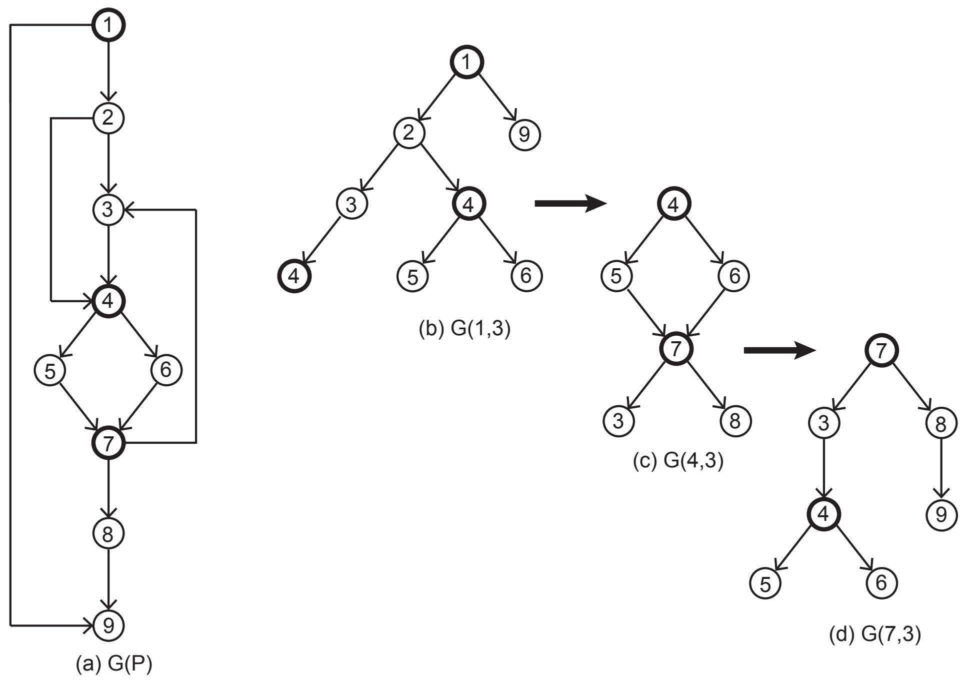

FIGURE 5.4: Partition a control flow graph (a) into 3 components (b), (c) and (d).

In addition, there are two special cases in partitioning a CFG that need to be considered. One is that the tree $\mathsf{G}(x, k)$ may cover a segment of the target program involving such computations that are more complex in control structure and/or data processing. This indicates that the state space of the corresponding (behavioral) model may be too large to be explored efficiently by means of online model checking. To reduce the state space of the (behavioral) model, a possible solution is to cut $\mathsf{G}(x, k)$ vertically following the dashed line or horizontally following the dotted line as illustrated in Fig. 5.5 (a). Cutting $\mathsf{G}(x, k)$ vertically results in two (partial) models, which need to be checked currently whenever the monitoring point x is reached. Cutting $\mathsf{G}(x, k)$ horizontally results in a k'-bounded tree $\mathsf{G}(x, k')$ with $k' < k$. In fact, there is no need to make all the partitions of $\mathsf{G}(\mathsf{P})$ k-bounded. We keep the bound k constant just for the sake of simplicity.

The other is that the length of a control loop in $\mathsf{G}(\mathsf{P})$ may be too short, e.g., the *short* loop as illustrated in Fig. 5.5 (b). Obviously, the vertex x is the only monitoring point according to our partitioning algorithm. That is, the program state is monitored at each iteration of the loop during runtime. The shorter the loop, the higher the sample rate. To reduce the sample rate, a possible solution is to *syntactically* unfold the loop up to appropriate steps, say, 4 steps in this example. As a consequence, the state information can be monitored every 4 steps, instead of every 1 step. Of course, the memory footprint of the target program may be enlarged in this way.

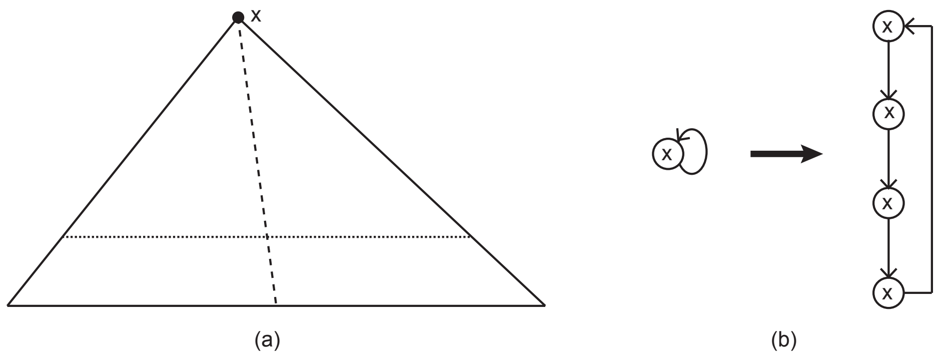

FIGURE 5.5: Two special cases in partitioning CFG

5.4 Summary

In this chapter, we first introduce a platform-independent programming abstraction for specifying such control software applications that exhibit time-periodic and multimodal features. This kind of software is usually safety-critical due to its use in the automotive, aerospace and other industries. A typical control application contains a set of periodic (software) tasks together with a mode-switching logic for activating or deactivating tasks. Our goal is to (online) check the correctness of the internal behaviors of the task that is identified as important among the others in the task set.

There are two ways to get the (behavioral) models at multiple levels of abstraction and/or from different perspectives for the program to be checked: one is to reuse the (design) models built during the software development phase; the other is to abstract the models directly from the source code of the target program. For the sake of simplicity, we suppose that the target program is written in a restricted subset of C that conforms to the (mandatory) requirements of the MISRA-C coding standard, which allows only static memory allocation and no function recursion.

The monitoring points for online model checking are determined by analyzing the control flow graph of the target program. We present a partitioning algorithm to calculate a smaller set of monitoring points that are distributed more or less evenly in the control flow graph. The time complexity of the algorithm is linear in the number of the vertices in the control flow graph.

Chapter 6

Integration of Online Model Checking with RTOS

As mentioned in Section 5.1, a typical embedded control application may switch from one operational mode to another during system execution. As a result, some tasks (in the current mode) are removed while others are added in the target mode. In order to ensure the correctness of the control software, in particular, the correctness of the tasks identified as safety-critical, during system execution, we present a framework for integrating the online model checking mechanism with the underlying real-time operating system (RTOS). As a proof of concept, a prototype is implemented by Krishna Sudhakar as his master thesis [84] cosupervised by the author. This work is later published in [85].

6.1 Integration Framework

Given a real-time operating system (RTOS), which manages one or more system applications running in a interleaved manner, we can deploy the online model checker as verification service inside the RTOS (with dashed lines) or outside the RTOS (without dashed lines) as illustrated in Fig. 6.1.

In both cases, an *observer* is needed to record the current state information while the system application under investigation is running. For a system application evaluated as safety-critical or ultra-reliable, we assume that its source code has already been instrumented with special system calls at the predetermined monitoring points in advance. Thus, the observer can be implemented inside the system call handler of this type of special system call. Whenever the special system call at some monitoring point is triggered, the observer (inside the system call handler) goes to read the values of the variables of interest at the current state, say s_i, out of the local memory of the target application, and then to apply the predefined mapping function to obtain the corresponding abstract state $\widehat{s_i} = \alpha(s_i)$ in the (behavioral) model of the target application. At this time, a simple *assertion* checking can be done if necessary. In case of violation, the operating system will be informed right away. Otherwise, $\widehat{s_i}$ will be written into the *ring buffer*[1], which is allocated by the operating system to store the current state information for online model checking. If the buffer is full, the oldest state will be overwritten by the latest state.

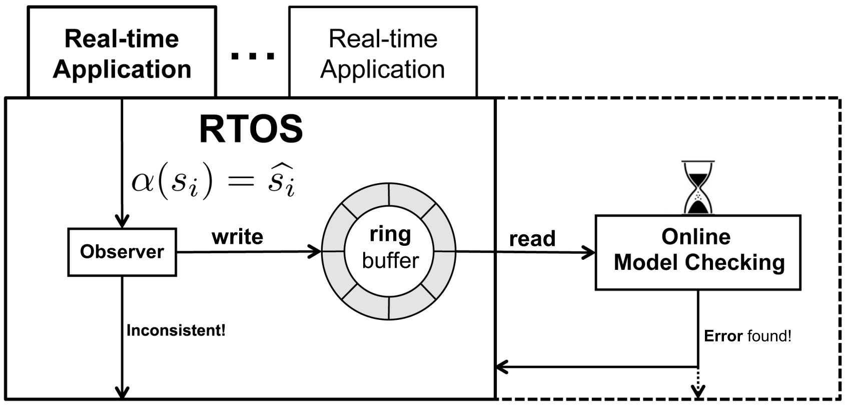

FIGURE 6.1: Integration Framework

On the other hand, the online model checker tries to take one (current) state out of the ring buffer every T time units, where T is the time limit allocated to the online model checker by the user. If the ring buffer is *empty*, the online model checker will either resume the work of the last checking cycle, provided that the work has not finished yet; or else simply wait until the next checking cycle starts. Otherwise, there is a state, say $\widehat{s_i}$, available, then within the given T time units, the online model checker tries to search for an error path of length up to k steps starting from $\widehat{s_i}$ in the (bounded) state space of the behavioral model of the target program. As a result, the following three possible checking results may be reported to the operating system:

Case *unsafe*: the checking process is finished in time, and an error path is found. In this case, an *alarm* will be sent to the operating system as quickly as possible. Notice that the error path might be *false negative*. However, to avoid the error really to happen, we have to conservatively choose to inform the operating system the potential danger. In response, the operating system may raise an *exception*. Considering that the exception handling is usually domain specific, thus we do not

[1]To reduce data misses, a (ring) buffer is used because the rate at which data is received and the rate at which it can be processed are variable over time.

discuss it here in a general sense. In addition, the error path can be recorded to help the user to figure out the cause of the error later.

Case *safe*: the checking process is finished in time, but no error path is found. That is, it is safe within the next k steps relative to $\widehat{s_i}$. The operating system continues its normal operation in this case.

Case *unkown*: the checking process is forcedly terminated due to *timeout*. Therefore, the state space of the (bounded) model is not exhaustively explored. It is, however, reasonable to believe that the probability is low of detecting errors near the current state $\widehat{s_i}$ in this case. It is up to the operating system to deal with this case anyway.

Needless to say, monitoring state information during program execution has more or less an influence on the performance of the target program as well as the underlying operating system. By introducing a special type of system call and its system call handler to fulfill the monitoring work, both the source code of the target application and the operating system need not to be modified too much. Since the system call handler usually has to consume some processing time, this in turn will limit the number of variables to be monitored. In this case, only the most important variables are selected to be monitored during runtime.

A variable is considered important, if it is used in a conditional statement of the target program. That is, the value of such a variable may make a contribution in deciding the control flow of the program's execution at the control point, at which multiple branches exist. Accordingly, the state space of the program model to be explored by the online model checker is reduced, because only one branch, instead of all the branches, of the program at this point needs to be taken into account.

Notice that the state information monitored for online model checking is used to reduce the state space of the behavioral model to be searched. In order to decrease the monitoring overhead by monitoring a subset of the state variables, this incomplete information may not uniquely identify an individual state, but a set of states in the behavioral model (see Section 4.1.4). As a consequence, the online model checker maybe needs to search in a larger state space of the behavioral model. The checking result is, however, not affected as long as the checking time is sufficient. This is different from online monitoring whereby the analysis may still not be accurate no matter how much time is taken due to the incomplete information collected during runtime.

In addition, the schedulability analysis for the target system with online model checking integrated can be conducted offline beforehand. Because the locations of the monitoring points and the number of the variables to be monitored are known in advance, it is thus possible to estimate the monitoring overhead and then statically analyze the worst-case execution time (WCET) of the source code of the system application during runtime.

By now we've explained the integration framework in a general sense at the conceptual level. The development of embedded systems indicates a trend towards incorporating more than one CPU, i.e., maybe multiple cores on a chip or multiple chips on a board or any combination thereof [86]. For example, in the automotive industry, the AUTOSAR OS specification published in 2011 added support for multicore systems [87]. The integration framework can then be implemented on different hardware architectures from single-core or multicore processor to multiprocessor:

- **Single Core Processor:** One possible implementation is to make the online model checker as a system service in the RTOS kernel [88]. In doing so, a fixed time slot is reserved a priori for online model checking, say, at the beginning (or end) of each scheduling cycle of the RTOS. This time slot is specifically reserved for the (online) verification service. If no online checking task is active, the scheduler is allowed to allocate this slot to such preemptive low priority tasks that can be moved or replaced by the online checking task at any time when the verification service is triggered. The advantage of this way of integration is that the communication overhead between the RTOS and the online model checker is very low. However, including the online model checker in the kernel space increases the footprint of the RTOS, even in the cases that the online checking service is not requested. In addition, the online model checker is usually a computationally intensive task, which requires additional memory and consumes additional energy. Another possible solution is to introduce a hypervisor to build multiple virtual machines, and then make the online model checker and the RTOS run on different virtual machines. Of course, a hypervisor will play a more valuable role on a multicore platform [86].
- **Multicore Processor:** From the software perspective, there are basically two types of multicore designs: Asymmetric Multi-Processing (AMP) and Symmetric Multi-Processing (SMP) [86, 89, 90]. In both cases, the online model checker can be assigned to a (specific) core different from the one that runs the RTOS. In this way, the memory footprint of the RTOS is reduced by deploying the online model checker outside the RTOS kernel. This also does not affect schedulability so much, as the only special additions to the RTOS are an "observer" and a function to communicate with the online model checker. However, for a multicore platform, special attention must be paid to ensure that the software running on different cores, be it operating systems or applications, are properly separated from each other so that they do not interfere with each other. Fortunately, there are different

techniques, in particular, different virtualization techniques, such as container and hypervisor [67], to fulfill this goal.

- **Multiprocessor:** The processors in a multiprocessor system may be tightly coupled at the bus level or loosely coupled via Internet for communication. The online model checker and the RTOS can thus be deployed on different processors (or nodes). They communicate with each other via I/O ports. Of course, correct delivery of messages needs to be guaranteed. In this way, the task(s) running on the RTOS can be verified by the online model checker on a remote machine. Theoretically, it is possible to verify distributed (real-time) applications by means of online model checking. Since the RTOS and the online model checker are deployed over a network, a number of factors comes into play when calculating the delay in communication, such as, bandwidth, error rate, noise, and so on.

Which solution is the best choice? That depends on many factors, such as the workload of the online model checker, the (real-time) operating system, the communication overhead, and the performance of the underlying hardware, to name just a few.

6.2 ORCOS

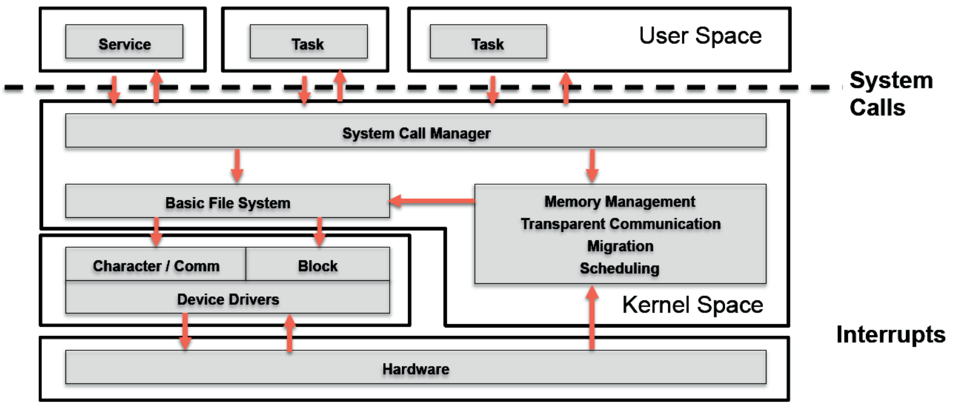

FIGURE 6.2: Architecture of ORCOS [91]

Organic ReConfigurable Operating System (ORCOS) [91] is a small-footprint real-time operating system[2] designed to be configurable during design-time and even during runtime [92]. Through a special configuration language based on XML, the user is able to configure only the functionality which is actually needed and decide which functionality to place in kernel and which in userspace [93]. The architecture of ORCOS is illustrated in Fig. 6.2. ORCOS is implemented using fully object oriented programming with C++

[2]Developed at the University of Paderborn in the research group chaired by Prof. Franz Rammig.

and suitable for most of the processors available to embedded systems, e.g., PowerPC405, Sparc Leon3, ARMv4(t) and above, QEMU (emulating PowerPC405), OMAP3530 SOC (Limited Support), etc..

The kernel of ORCOS is made up of several modules, which can be configured through the XML-based configuration language SCL (Skeleton Customization Language) [94]. Now let's break it down.

Processes The processes are called tasks in ORCOS. A task is just a resource container, its executing entity is called thread. A thread is the unit of execution and scheduling, not the task. In order to guarantee predictability, ORCOS introduces a special type of task called worker task. In fact, there is only one worker task in ORCOS. This worker task belongs to the kernel space and can spawn multiple worker threads. Each worker thread has its own stack and shares the resource of the work task. The worker threads can take over such kind of work that arrives non-deterministically, e.g., asynchronous IO interrupts, or that needs to be executed at a specific time, e.g., timed calls or periodic calls to functions. The introduction of the worker threads allows these activities to be scheduled like any other threads [95].

Scheduler The scheduler is in the core of the ORCOS kernel. ORCOS schedules based on threads. The following scheduling strategies are implemented in ORCOS: Round Robin (RR), Fixed-Priority Scheduling (FPS), Rate Monotonic Scheduling (RMS) [96], Earliest Deadline First (EDF) [96] and EDF with Total Bandwidth Server (TBS) [97].

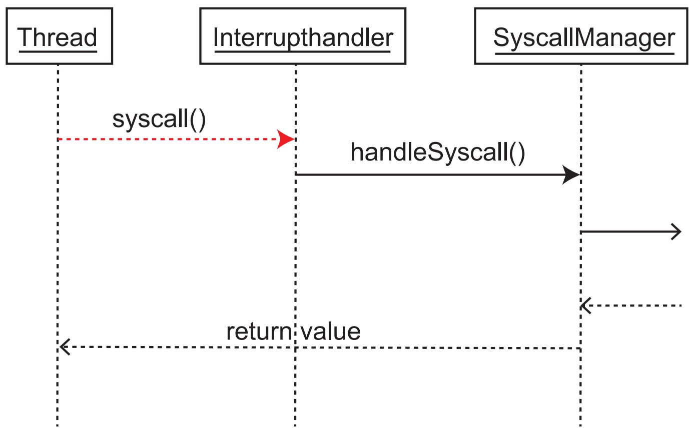

FIGURE 6.3: Syscall processing of ORCOS [91]

System Calls The tasks as well as services in user space can use kernel functionalities only through the system calls defined in the syscall API. The processing of a system call is illustrated in Fig. 6.3. Whenever a thread issues a system call, the related information of the system call will be stored at the specific locations on the stack of the task. Afterwards, a software interrupt is raised and then the kernel will take control, which involves storing the context of the task and triggering an appropriate interrupt handler. Thereafter, the `SyscallManager` will be invoked: the parameters as well as the syscall number are loaded from the stack of the task and the functionality of the system call is executed. Notice that there is no direct function call on the `Interrupthandler`. Instead, the `syscall()` will cause a sequence of assembler instructions to be executed, which react on the software interrupt and end up calling the `Interrupthandler`.

Communication ORCOS allows inter-node and inter-process communication. Two processes or nodes communicate with each other using socket API. The socket design is implemented using a configurable protocol stack. Each socket can be explicitly configured (even at runtime) to define which protocol stack will be used.

Memory Management The memory management belongs to the core of the ORCOS kernel. ORCOS follows the rule of separating the kernel and every task's memory from each other. Each task has its own memory manager. In addition, ORCOS is capable of virtual addressing if the underlying hardware architecture is equiped with a Memory Management Unit (MMU).

Filesystem The filesystem of ORCOS is inspired by the Unix filesystem. Whenever a resource is created, it will automatically register itself at the `Filemanager`. Each device is accessible by a unique path.

HAL The ORCOS kernel does not operate directly on the hardware in order to maintain portability to other hardware platforms. Instead, the kernel delegates the calls to the hardware through a Hardware Abstraction Layer (HAL), which offers an interface independent of any hardware platform, which in turn delegates the calls to the real underlying hardware.

Power Management ORCOS has built-in power management to control the power consumption of the devices that support throttling or other power states through their device drivers.

6.3 Prototype Implementation

As a proof of concept, we've integrated the online model checker with ORCOS on top of a (virtualized) multicore platform. Fig. 6.4 illustrates the architecture of our prototype implementation. The whole system is built on a Linux platform (32 bit Ubuntu 12.04 LTS) with 3GHz Pentium 4 CPU and 2GB RAM. ORCOS runs on top of QEMU [98], a open source machine emulator that emulates in our setting the microcontroller PowerPC 405EP with 333MHz CPU and 4KB on-chip memory. During the time when ORCOS was developing, QEMU v1.5.0 was the highest version available for download. This version of QEMU fails to emulate the networking capability of PowerPC 405EP, which makes ORCOS impossible to communicate with the outside world. Therefore, a communication helper[3] is used to connect ORCOS with the outside world. The online model checker is implemented as verification service on Linux because it adopts a tailored zChaff SAT solver as its search engine, which needs to run on a general purpose operating system.

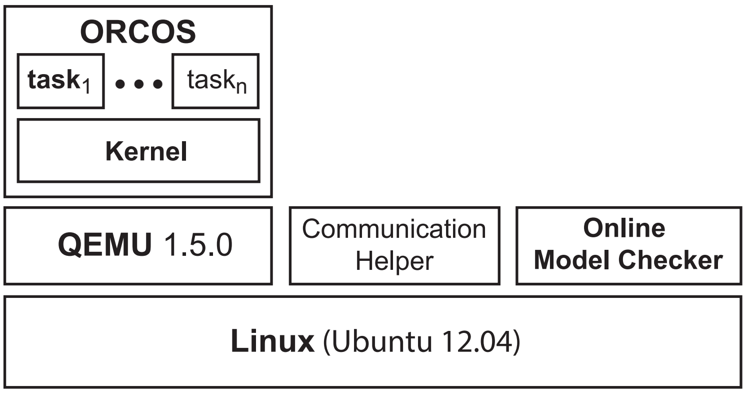

FIGURE 6.4: Architecture of prototype implementation

In some sense, this implementation comes close to a productive system running on a multicore platform (as shown in Fig. 4.17 in Section 4.2.3), where the software application on top of ORCOS and the online model checker on top of Linux run on different virtual machines. In other words, ORCOS on top of QEMU (emulating PowerPC 405EP) is running as a process on the host, i.e., the Linux platform. The online model checker is running as another process on the same host. Notice that both processes can be seen as an own virtual machine on top of a (virtualized) multi-core platform. In this sense, the Linux platform also plays a role of hypervisor. The communication helper emulates then a standard inter-VM (Virtual Machine) communication service of the hypervisor.

Fig. 6.5 illustrates the details of our implementation. In order to probe the state information during runtime, we add a special system call `monitor_read()`, an `observer`, a

[3]A plugin to QEMU developed in Java by Sijia Li, a student assistant in the research group chaired by Prof. Franz Rammig.

`ring buffer` and a special `worker thread` into the kernel of ORCOS. For a task to be checked, we assume that the `monitor_read()` system calls have already been inserted at the monitoring points in the source code of the task at the software development phase. The system call `monitor_read()` is handled by the `observer` inside `SyscallManager`. The `observer` is a snippet of code that copies the values of the variables being monitored from the memory of the task into the `ring buffer` in the kernel space. The `ring buffer` occupies a contiguous memory region, whose structure is decided by the number and types of the variables to be monitored. The `worker thread` functions as a `delegate` of the online model checker inside ORCOS. If the `ring buffer` is *not* empty, the `delegate` does periodically the following work: it takes a (concrete) state from the `ring buffer`, maps this state into the corresponding abstract state, and then delivers the abstract state to the online model checker.

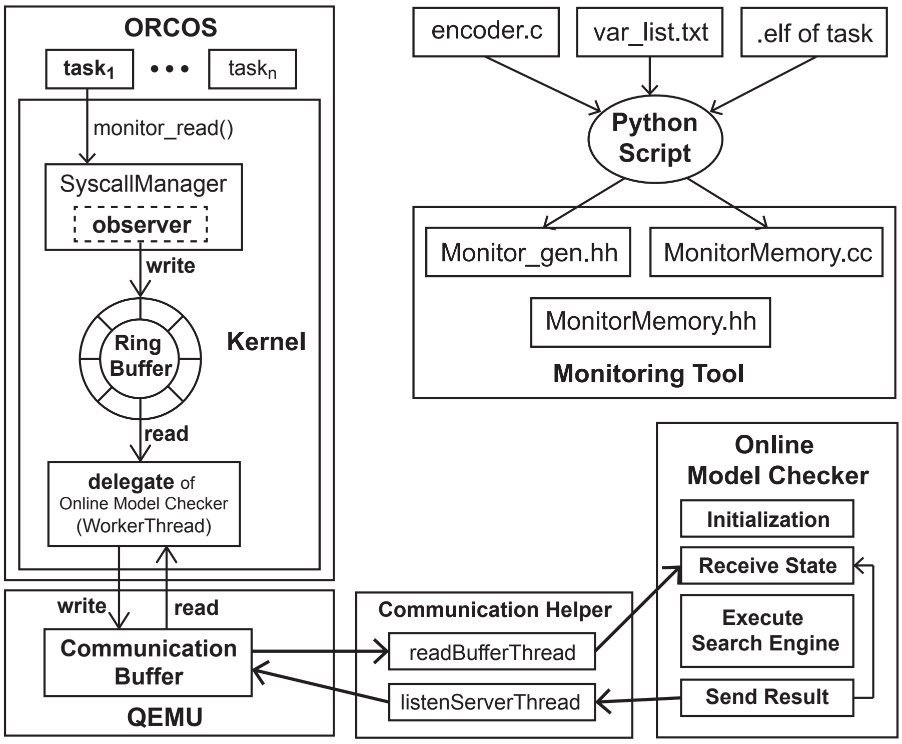

FIGURE 6.5: Integration of online model checking with ORCOS

Monitoring Tool

A Python script is used to generate the monitoring tool for ORCOS. The Python script takes three files (provided by the user) as input: `var_list.txt`, `encoder.c` and the `.elf` file of the task to be checked. The file `var_list.txt` contains the variables to be monitored. The file `encoder.c` contains the mapping functions. Each mapping function defines as a predicate, which accepts the variables in `var_list.txt` as parameters and returns a *Boolean* value. By disassembling the `.elf` file of the target program, we are able to obtain the detail information about the variables of interest, such as data type, memory address, and so on. Notice that the program is assumed to be written in C and compliant to the MISRA C standard (see Section 5.3.1). In addition, for the sake of simplicity we also assume that the program does not call such external function(s) whose source code (in C) is not available. Because neither dynamic memory allocation nor function recursion is allowed, the correspondence of variables to addresses can thus be established once and for all at compile time [28].

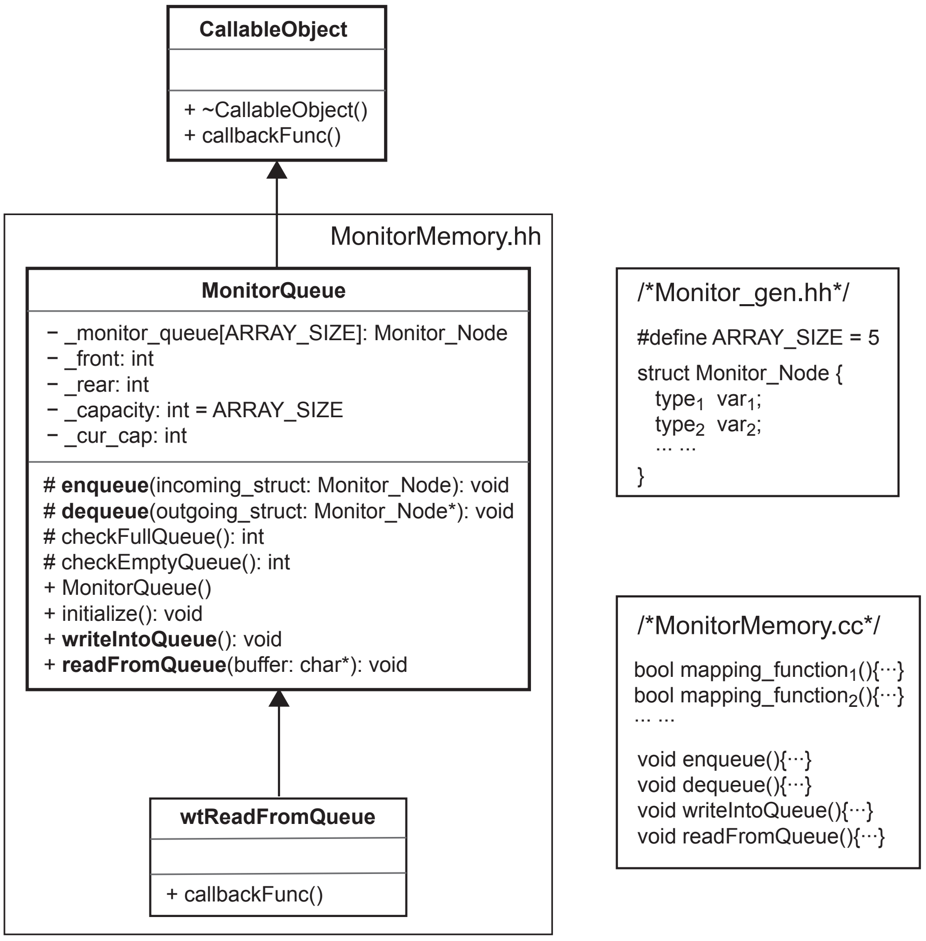

FIGURE 6.6: Monitoring tool of ORCOS

The Python script produces two files `Monitor_gen.hh` and `MonitorMemory.cc` as output. These two files together with `MonitorMemory.hh` build the monitoring tool of ORCOS as shown in Fig. 6.6. The header file `Monitor_gen.hh` contains the structure definition of `Monitor_Node` and the macro definition `ARRAY_SIZE`. The former defines a row of the `ring buffer`, and the latter the length of the `ring buffer`. The header file `MonitorMemory.hh` contains `class MonitorQueue`, which defines the `ring buffer` (i.e., `monitor_queue`) and its interface. This file is not generated by the Python script, because the definition of `class MonitorQueue` is fixed. The `ring buffer` is declared as a `static` member, so that all the instances of `class MonitorQueue` and its subclasses can operate on the same `ring buffer`. The source file `MonitorMemory.cc` contains the mapping functions given in `encoder.c` as well as the definitions of the four interface functions of the `ring buffer`: `enqueue()`, `dequeue()`, `writeIntoQueue()` and `readFromQueue()`.

The access to `enqueue()` and `dequeue()` is protected, while `writeIntoQueue()` and `readFromQueue()` are publicly accessible. `enqueue()` adds one state entry to the end of the `ring buffer`. When the buffer is full, the oldest state entry will be overwritten by the newest one. `dequeue()` removes one state entry from the `ring buffer` and stores it into a temporary location. `writeIntoQueue()` copies the current state entry from the local memory of the task being checked in the user space into a temporary location in the kernel space, and then puts it into the `ring buffer` by `enqueue()`.

Worker Thread

ORCOS has to exchange information with the outside world through the `communication buffer` shared with QEMU. A specific worker thread is used to transfer the data from the ring buffer to the communication buffer and then send it to the online model checker. The priority of the worker thread is set to lower than that of the task being checked. The worker thread is an instance of `class wtReadFromQueue`. As illustrated in Fig. 6.6, `class wtReadFromQueue` inherits `class MonitorQueue` in `MonitorMemory.hh`. It has no member data but one member function `callbackFunc()`. This function is the `main` function of the worker thread.

The worker thread calls `readFromQueue()` to take a (concrete) state from the `ring buffer` by `dequeue()`, applies the mapping functions to this state to get an abstract one, and then puts the abstract state into the `communication buffer` using `sendwithQEMU()`. After the online model checker sends the checking result back to the `communication buffer`, the worker thread is also responsible for fetching the checking result from the `communication buffer` using `receivefrom()`. In case that the result indicates unsafe, the worker thread then tries to inform ORCOS as quickly as possible. `sendwithQEMU()` and `receivefrom()` are member functions of `class qemueth`, which is implemented inside ORCOS. `sendwithQEMU()` writes the data into the `communication buffer`, while `receivefrom()` reads the data out of the `communication buffer`.

The worker thread can be seen as a delegate of the online model checker inside ORCOS.

Communication Helper

The `communication helper` acts as an intermediary between ORCOS and the outside world for exchanging data. It is implemented in Java as plugin to QEMU. There are mainly two threads `readBufferThread` and `listenSeverThread`. `readBufferThread` reads a data out of the `communication buffer` and then sends it to a connected sever using the TCP/IP protocol. `listenSeverThread` receives a data from a connected sever using the TCP/IP protocol and then writes it into the `communication buffer`.

Online Model Checker

The online model checker can be seen as a server application, which is implemented as a process of the Linux platform. We use a tailored zChaff SAT solver (see Section 4.2.4) as its search engine. The (bounded) behavioral model of the task as well as the property to be checked is encoded as CNF formula in DIMACS format. At the initialization phase, the online model checker loads the model together with the property into memory. Once the TCP/IP connection is established between the communication helper and the online model checker, it waits for a state information sent from ORCOS. Upon receiving the state information, it goes to search the state space of the behavioral model for an error by the SAT solver within a predefined time limit. By timeout it terminates the search anyway and sends the result (i.e., safe, unsafe, or unknown) back to ORCOS. Thereafter the online model checker waits for a new state information and repeats the above steps until an error is found or the task being checked terminates.

6.4 Evaluation

The integration of online model checking with RTOS introduces unavoidably additional overhead for monitoring the state information and transferring the data between the RTOS and the online model checker. We first analyze qualitatively the constitution of the overhead to figure out the factors that may affect the performance of the RTOS as well as the task under investigation during runtime. Afterwards, we provide a quantitive measurement of the monitoring overhead and the communication overhead.

Although the Linux platform can not exactly reflect a real multicore system and QEMU can not exactly emulate the target hardware as it is, it is still meaningful to implement our online model checking mechanism on this virtual multicore platform. The implementation of the observer and the online model checker is mainly dependent on the respective operating system. If the whole system is ported to a real hardware platform, the only part of the implementation that needs to be modified is the code segment responsible for the communication between the observer (via the RTOS) and the online model checker.

The experimental results are somewhat biased by our implementation using an emulator. Therefore, the relative results (i.e., the time consumption as a function of bytes to be processed) are more meaningful than the absolute values. To our knowledge, there should exist a quasi linear relationship between the actual values and the measured results. In this sense, a reasonable order of magnitude, to which the actual values may belong, can be estimated based on the results measured on the emulator.

6.4.1 Overhead Analysis

Here we give an analytical estimation of the overhead introduced by the online model checking mechanism, i.e., the monitoring overhead and the communication overhead. The analysis aims to determine the factors that can help to reduce the overhead.

Monitoring Overhead

Without loss of generality, let $task_i$ be a safety critical task to be checked. Then, the total execution time τ_i of $task_i$ holds

$$\tau_i \propto \mathbf{T}_{task} + (\mathbf{T}_{syscall} \times \mathbf{N}_{syscall}), \text{for } \mathbf{N}_{syscall} \geq 1,$$

where $\mathbf{T}_{task}$ is the WCET of $task_i$ without taking into account the online model checking, $\mathbf{T}_{syscall}$ is the processing time of the system call for monitoring variables, and $\mathbf{N}_{syscall}$ is the number of the special system calls (i.e., `monitor_read()`) in $task_i$.

$\mathbf{T}_{syscall}$ is proportional to the number $\mathbf{N}_{byte}$ of the bytes being copied, together with some constant processing time of `syscall`, i.e., context switch, interrupt handling, etc., thus,

$$\mathbf{T}_{syscall} \propto \mathbf{T}_{copy\mathbf{N}_{byte}} + \mathbf{C}_{syscall}.$$

Therefore, the total execution time τ_i is defined as

$$\tau_i = \mathbf{T}_{task} + (\mathbf{T}_{copy\mathbf{N}_{byte}} + \mathbf{C}_{syscall}) \times \mathbf{N}_{syscall}.$$

As a result, the monitoring overhead is calculated as

$$\mathbf{P}_{overhead} = \frac{\mathbf{T}_{syscall} \times \mathbf{N}_{syscall}}{\mathbf{T}_{task}}.$$

The monitoring overhead is proportional to the number of the system calls for monitoring variables, provided that the number of the variables to be monitored is fixed. In order to reduce the monitoring overhead, it is better to set the normal system calls as monitoring points as long as possible. In this way, $\mathbf{C}_{syscall}$ needs not to be counted as a part of the monitoring overhead, because these system calls belong to the normal behaviors of the task. Now let's define $\mathbf{N}_{syscall} = \mathbf{N}'_{syscall} + \mathbf{N}''_{syscall}$, where $\mathbf{N}'_{syscall}$ is the number of the special system calls and $\mathbf{N}''_{syscall}$ the number of the normal system calls but set as monitoring points. Then the monitoring overhead is defined as

$$\mathbf{P}_{overhead} = \frac{\mathbf{T}_{syscall} \times \mathbf{N}'_{syscall} + \mathbf{T}_{copy\mathbf{N}_{byte}} \times \mathbf{N}''_{syscall}}{\mathbf{T}_{task}}.$$

In particular, if all the monitoring points can be set in those places in which the normal system calls locate, i.e., $\mathbf{N}'_{syscall} = 0$ and $\mathbf{N}_{syscall} = \mathbf{N}''_{syscall}$, then, the monitoring overhead is reduced to be

$$\mathbf{P}_{overhead} = \frac{\mathbf{T}_{copy\mathbf{N}_{byte}} \times \mathbf{N}_{syscall}}{\mathbf{T}_{task}}.$$

Recall that the monitoring points are predetermined at the software development phase, the only way to reduce the monitoring overhead is to reduce the number of the variables to be monitored. For this purpose, it is better associate a weight to each variable by some criteria such that the larger the weight of a variable, the more impact the variable has on deciding the paths in the behavioral model of the program to be checked. However, the incomplete state information may increase the workload of the online model checker (see Section 4.1.4).

Communication Overhead

Recall that the communication between ORCOS and the online model checker is implemented by a specific worker thread. For the sake of simplicity, the worker thread acting as the delegate of the online model checker is called *delegate* in the sequel. The main work of the delegate is to read the data from the ring buffer, apply it to the mapping functions, and then send the results to the online model checker. In addition, it also deals with the checking result sent back from the online model checker. Therefore, the execution time τ_{wt} of the delegate is defined as

$$\tau_{wt} = \mathbf{T}_{comm} + \mathbf{C},$$

where $\mathbf{C}$ is a constant time taken for context switches and for executing the remainder code of the delegate.

The time $\mathbf{T}_{comm}$ taken for transferring the data between ORCOS and the online model checker is made up of the following three factors:

- $\mathbf{T}_{ORCOS\_to\_olmc}$: time taken for transferring the data from ORCOS to the online model checker;
- $\mathbf{C}_{olmc}$: constant time taken for running the online model checker; and
- $\mathbf{T}_{olmc\_to\_ORCOS}$: time taken for transferring the data from the online model checker back to ORCOS.

It is worth pointing out that $\mathbf{T}_{ORCOS\_to\_olmc}$ and $\mathbf{T}_{olmc\_to\_ORCOS}$ are determined mainly by how the data exchange between ORCOS and the online model checker is implemented on what kind of system architecture. Given a real multi-core platform, $\mathbf{T}_{ORCOS\_to\_olmc}$ and $\mathbf{T}_{olmc\_to\_ORCOS}$ should be much smaller than our implementation on top of QEMU together with the `communication helper`.

$\mathbf{T}_{ORCOS\_to\_olmc}$ depends on $\mathbf{T}_{ORCOS\_to\_QEMU}$, $\mathbf{T}_{QEMU\_to\_helper}$ and $\mathbf{T}_{helper\_to\_olmc}$ as illustrated in Fig. 6.7, where $\mathbf{T}_{ORCOS\_to\_QEMU}$ is the time taken for writing the data into the `communication buffer`; $\mathbf{T}_{QEMU\_to\_helper}$ is the time taken for reading the data out of the `communication buffer`; and $\mathbf{T}_{helper\_to\_olmc}$ is the time taken for sending the data from the communication helper to the online model checker using TCP/IP protocol. In addition, there is also a constant time taken for sending an ACK from the online model checker back to the communication helper. Therefore,

$$\mathbf{T}_{ORCOS\_to\_olmc} = \mathbf{T}_{ORCOS\_to\_QEMU} + \mathbf{T}_{QEMU\_to\_helper} + \mathbf{T}_{helper\_to\_olmc} + \mathbf{C},$$

where $\mathbf{C}$ is a constant time taken for context switches, receiving ACKs, etc..

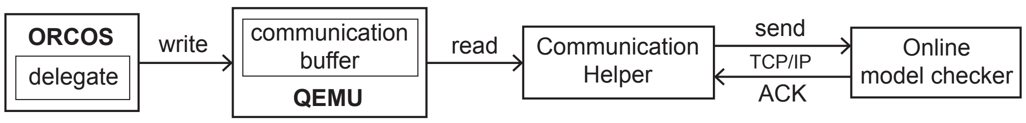

FIGURE 6.7: Sending state information to online model checker

The time $\mathbf{T}_{ORCOS\_to\_QEMU}$ consists mainly of the following three factors:

- $\mathbf{T}_{deque}$: time taken for copying a (concrete) state out of the `ring buffer` inside ORCOS;
- $\mathbf{T}_{map}$: time taken for mapping the concrete state into the corresponding abstract state according to the given mapping functions; and

- $\mathbf{T}_{copy\_to\_buf}$: time taken for copying the (abstract) state to the `communication buffer` inside QEMU.

Further, $\mathbf{T}_{deque}$ is proportional to the size of the data being monitored; $\mathbf{T}_{map}$ depends on the number of the mapping functions to be applied and the execution time of each mapping function; $\mathbf{T}_{copy\_to\_buf}$ is proportional to the number of the mapping functions, because the return values of the mapping functions are the values being copied to the `communication buffer`. Therefore, we have

$$\begin{aligned} \mathbf{T}_{deque} &\propto \mathbf{N}_{byte}, \\ \mathbf{T}_{map} &\propto \mathbf{N}_{map\_func} \times \mathbf{T}_{map\_func}, \text{ and} \\ \mathbf{T}_{copy\_to\_buf} &\propto \mathbf{N}_{map\_func}. \end{aligned}$$

The computation time $\mathbf{C}_{olmc}$ of the online model checker is considered to be constant, because the online model checker is always assigned by the user a predefined time limit for its execution. In case of timeout it must terminate its execution anyway.

$\mathbf{T}_{olmc\_to\_ORCOS}$ depends on $\mathbf{T}_{olmc\_to\_helper}$, $\mathbf{T}_{helper\_to\_QEMU}$, and $\mathbf{T}_{QEMU\_to\_ORCOS}$ as illustrated in Fig. 6.8. Here $\mathbf{T}_{olmc\_to\_helper}$ is the time taken for sending the checking result from the online model checker to the communication helper through the TCP/IP connection. In addition, there is also a constant time taken for sending an ACK from the communication helper back to the online model checker. $\mathbf{T}_{helper\_to\_QEMU}$ is the time taken for writing the checking result into the `communication buffer`. $\mathbf{T}_{QEMU\_to\_ORCOS}$ is the time taken for reading the checking result from the `communication buffer`. Therefore,

$$\mathbf{T}_{olmc\_to\_ORCOS} = \mathbf{T}_{olmc\_to\_helper} + \mathbf{T}_{helper\_to\_QEMU} + \mathbf{T}_{QEMU\_to\_ORCOS} + \mathbf{C},$$

where $\mathbf{C}$ is a constant time taken for context switches, sending ACKs, etc..

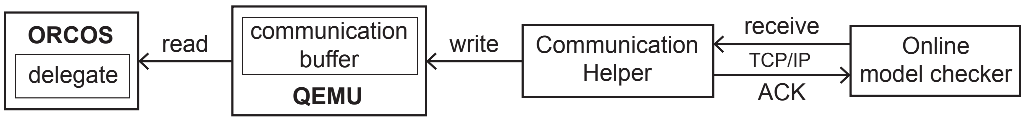

FIGURE 6.8: Receiving checking result from online model checker

Notice that the communication helper is in fact transparent to ORCOS and to the online model checker. In order to send a data to the online model checker, what the delegate needs to do is to put the data into the `communication buffer`; on the other hand, the online model checker connects to a certain port and listen for the incoming connections.

There are two ways to synchronize the communication between the delegate/ORCOS and the online model checker: *synchronous* and *asynchronous* communication.

Synchronous Communication As illustrated on the left hand side of Fig. 6.9, after sending the data to the online model checker, the delegate needs to wait for the checking result sent back by the online model checker in the current execution cycle. This means that the delegate has to enter the *blocked* state until the checking result has arrived. If the delegate is set to the highest priority, it will move directly to the running state and then preempt the currently running task. In case of a negative result, the alarm will be sent to ORCOS by the *same* execution instance of the delegate as quickly as possible.

The WCET taken for synchronous communication is thus defined as

$$\mathbf{T}_{comm} \quad = \quad \mathbf{T}_{ORCOS\_to\_olmc} + \mathbf{C}_{olmc} + \mathbf{T}_{olmc\_to\_ORCOS} + \mathbf{C}$$

where **C** is a constant time taken for context switches, sending ACKs, etc..

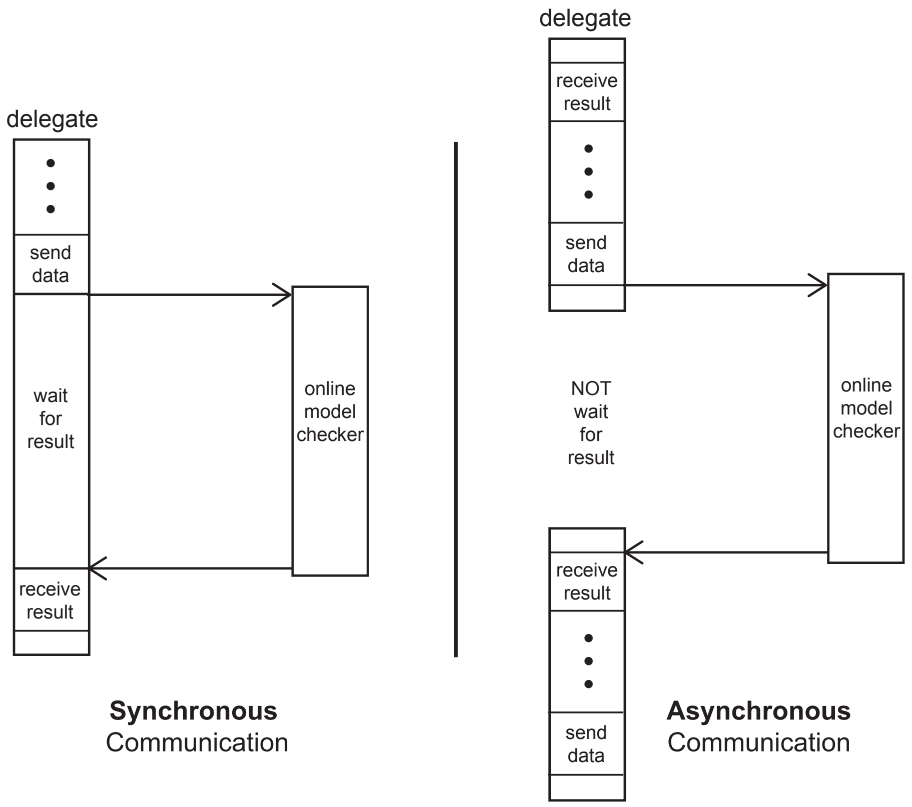

FIGURE 6.9: Synchronous and Asynchronous Communication

Asynchronous Communication As illustrated on the right hand side of Fig. 6.9, after sending the data to the online model checker, the delegate does not need to wait for the checking result sent by the online model checker in the current execution cycle. Instead, the result is checked by a successive execution instance of the delegate. That is, in each execution cycle, the delegate first checks the `communication buffer` for the result from the online model checker with respect to the data sent in a preceding execution cycle. If the result does not indicate an error, the delegate will send the new data to the online model checker; otherwise, the delegate will inform ORCOS of the potential error and then terminate its execution. Therefore, in case of a negative result, the alarm will be sent to ORCOS in a *successive* execution instance of the delegate.

The WCET taken for asynchronous communication is thus defined as

$$\mathbf{T}_{comm} = \begin{cases} \mathbf{T}_{olmc\_to\_ORCOS} + \mathbf{C}, & \text{error} \\ \mathbf{T}_{olmc\_to\_ORCOS} + \mathbf{T}_{ORCOS\_to\_olmc} + \mathbf{C}, & \text{otherwise} \end{cases}$$

where **C** is a constant time taken for context switches, sending ACKs, etc..

Given the behavioral model to be checked, the number $\mathbf{N}_{map\_func}$ of the mapping functions is usually fixed. Consequently, the time $\mathbf{T}_{deque}$, $\mathbf{T}_{map}$ and $\mathbf{T}_{copy\_to\_buf}$ can not be reduced. Notice that the checking result of the online model checker can be configured to send back to ORCOS only when a violation against the property is detected. In this way, $\mathbf{T}_{olmc\_to\_ORCOS}$ can be neglected, provided that the errors (if any) in the target program are very few. This assumption is reasonable because a safety-critical program should have been intensively checked before it is released. The only way to reduce the execution time τ_{wt} of the delegate is to reduce the transmission time from ORCOS to the online model checker, i.e., $\mathbf{T}_{ORCOS\_to\_olmc}$.

6.4.2 Overhead Measurement

Based on our prototype implementation we've measured the execution time taken for monitoring the variables (i.e., $\mathbf{T}_{syscall}$) and for transferring the data between ORCOS and the online model checker (i.e., $\mathbf{T}_{ORCOS\_to\_olmc}$ and $\mathbf{T}_{olmc\_to\_ORCOS}$). We have to point out that the QEMU just emulates the target Instruction Set Architecture (ISA) as well as the hardware interface of the peripheral devices [99]. That is, no timing model of the target architecture is implemented in QEMU. The execution time measured depends thus not only on the implementation of QEMU but also on the configuration of the host computer, i.e., the type of the CPU, the size of the memory, the scheduling algorithm and the workload of the host operating system, to name just a few. As a consequence, the measured results do not exactly reflect the actual overhead produced on the native PowerPC microcontroller for online model checking. In this sense, the relative results (i.e., the time consumption as a function of bytes to be processed) are more meaningful than the absolute values. In addition, the measured results do provide us with a clue of the order of magnitude, of which the actual values may be. Our primary goal is to determine those factors that may affect the performance of the online model checking so as to improve the online model checking mechanism.

Monitoring State Information

We've instrumented the source code of the task to be checked with the `monitor_read()` system calls in advance. Whenever `monitor_read()` is executed, the "observer" (embedded in the system call manager) will copy the values of the variables of interest from the memory of the task to the `ring buffer` in the kernel space. We've measured the time taken for copying the data of different size from 10 bytes up to 100 bytes stepping increasingly by 10 bytes. For each setting, the measurement is done 50 times in order to obtain an average value. The time is measured in microseconds by getting the difference of the time from the start of the system call to its end.

Fig. 6.10 shows the average time taken for monitoring the state information in different size settings together with the corresponding linear regression line defined by the linear equation $f(x) = 8.04x + 160.72$, which indicates that by increasing every 10 bytes, the time taken for monitoring increases approximately by $80.4\mu s$.

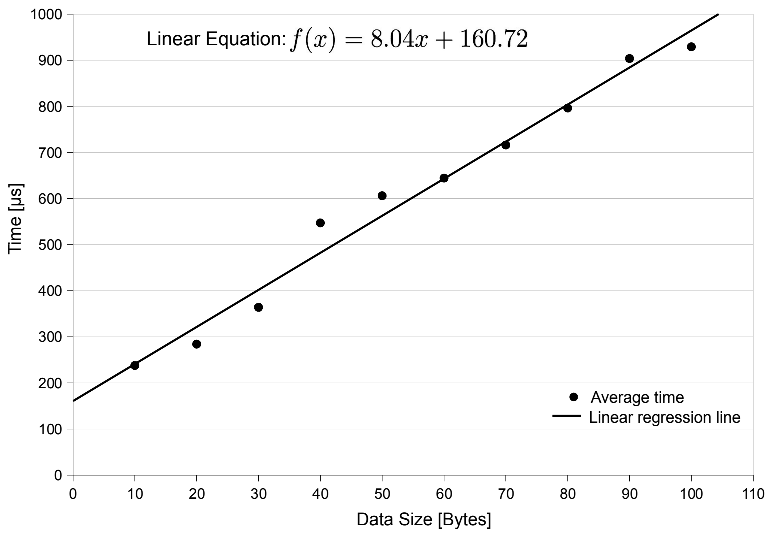

FIGURE 6.10: Time taken for monitoring state information

It is reasonable to believe that there should also exist a similar linear relationship between the size of the data and the time consumption in case that the measurement is done on a real hardware platform.

Sending State Information to Online Model Checker

We've measured the time $\mathbf{T}_{ORCOS\_to\_olmc}$ taken for transferring the data of different size from 10 bytes up to 100 bytes stepping increasingly by 10 bytes, too. The measurement is also done 50 times for each setting in order to obtain an average value. The time is measured in microseconds, which consists mainly of two parts: $\mathbf{T}_{ORCOS\_to\_QEMU}$ and $\mathbf{T}_{QEMU\_to\_olmc}$.

The transmission time $\mathbf{T}_{QEMU\_to\_olmc}$ from QEMU to the online model checker is almost constant, i.e., about $50\mu s$, for the data of different size from 10 bytes up to 100 bytes. The result is obtained by getting the difference of the time at which the communication helper begins to read the data from the (communication) buffer and the time at which the data is sent to the online model checker using the TCP/IP protocol. The constant transmission time lies in that the communication buffer is configured to accommodate the date of at least 100 bytes and the data is sent off to its destination in one TCP/IP packet. This transmission time depends largely on how the online model checker is integrated with RTOS. Shared memory seems to be the best solution whereby the time consumption can be neglected (to some degree).

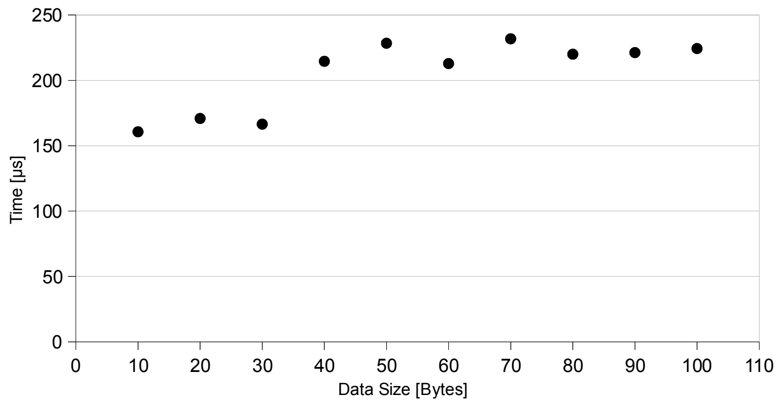

FIGURE 6.11: Time taken for transferring data from ORCOS buffer to QEMU buffer

The time $\mathbf{T}_{ORCOS\_to\_QEMU} = \mathbf{T}_{deque} + \mathbf{T}_{map} + \mathbf{T}_{copy\_to\_buf}$, i.e., it covers the time taken for reading the values (of the concrete state) from the ring buffer inside ORCOS, applying the mapping functions to them, and then writing the resulting values (of the abstract state) into the communication buffer inside QEMU. The measured time, as illustrated in Fig. 6.11, varies between $150\mu s$ and $200\mu s$ in average for the data of size from 10 bytes up to 30 bytes, while between $200\mu s$ and $250\mu s$ in average for the data of size from 40 bytes up to 100 bytes. Because the time $\mathbf{T}_{map}$ depends mainly on the complexity of the mapping functions, there should exist no linear relationship between the size of the data and the time consumption. Considering that the variables to be monitored and the mapping functions are given in advance, $\mathbf{T}_{ORCOS\_to\_QEMU}$ is hard to be reduced further.

Receiving Checking Result from Online Model Checker

The checking result is usually one byte[4] (say, 0 stands for safe, 1 for unsafe, and 2 for unknown). Therefore, the transmission time $\mathbf{T}_{olmc\_to\_ORCOS}$ can be seen as constant. The measurement is done 10 times in order to obtain an average result. The transmission time in average is about $372.82\mu s$. Although only one byte is sent back to ORCOS, $\mathbf{T}_{olmc\_to\_ORCOS}$ is somewhat greater than $\mathbf{T}_{ORCOS\_to\_olmc}$. The reason lies in that the delegate (i.e., the worker thread) may not be active at the time when the checking result is sent back to ORCOS. In this case, we have to wait until the delegate is activated by the scheduler of ORCOS and then the checking result will be processed. In practice, it is better to send the checking result back to ORCOS only when an error is detected. Thus, the checking result can be used as signal to trigger a software interrupt with a higher priority. In this way, the checking result can be processed earlier by ORCOS.

6.5 Discussion

In [12] four special requirements on the integration architecture are proposed for (online) monitoring fault tolerant real-time systems:

- *Functionality*: the monitor does not modify the nominal functionality of the target system, unless the target system itself violates the given (property) specification;
- *Schedulability*: the monitor does not interfere with the timeliness of the services provided by the target system, i.e., the monitoring mechanism does not cause the target system to violate its hard real-time guarantees, unless the target system itself violates the given (property) specification;
- *Reliability*: the monitoring mechanism does not decrease the (required) reliability of the target system, i.e., the reliability of the target system integrated with the monitor is greater than or equal to the reliability of the target system alone;
- *Certifiability*: the monitoring mechanism does not make the re-certification of the target system overly difficult, i.e., the monitoring mechanism does not add unduly modifications to the source code (or object code) of the target system.

If the above criteria are met, the real-time system will benefit by introducing a monitoring mechanism [12]. Obviously, the same goes for the integration of online model checking with RTOS. In our case, the source code of the target program is instrumented with predetermined finitely many monitoring points. The distance between any two adjacent monitoring points is not more than k (steps), a predefined bound for online (bounded) model checking. The state information is permitted to be probed only at these monitoring points. No other additional modifications to the source code are needed. Therefore,

[4]In case that an error is detected, the online model checker can provide more information if needed.

the functionality of the target system is not changed. Since the number of the monitoring points is fixed, so is the size of the state information to be monitored, as well as the time limit allocated to the online model checker, the additional monitoring overhead and communication overhead can be estimated a priori. Consequently, the schedulability of the target system can be decided ahead of time. The online model checker only needs to "read" the state information at monitoring points of the target program, and then informs the underlying operating system only when a violation is detected. Theoretically speaking, the reliability and the certifiability of the target system are not affected by online model checking.

6.6 Summary

First of all, we present a general framework for integration of online model checking with RTOS, such as ORCOS. This integration framework can be implemented on different hardware architectures from single-core or multicore processor to multiprocessor. As a proof of concept, we implement a prototype on top of a (virtualized) multicore platform. The implementation of the observer and the online model checker is mainly dependent on the respective operating system. If the whole system is ported to a real multicore platform, the only part of the implementation that needs to be modified is the code segment responsible for the communication between the observer (via the RTOS) and the online model checker.

Thereafter, we analyze qualitatively the constitution of the overhead to determine the factors that may affect the performance of the RTOS as well as the task to be checked and then provide a quantitive measurement of the monitoring overhead and the communication overhead. Although the experimental results can not exactly reflect the actual values produced on a native PowerPC microcontroller, the measured values do provide us with a clue of the order of magnitude, of which the actual values may be. According to our analysis, there are two ways to reduce the monitoring overhead: one is to set the native system calls in the source code as monitoring points as long as possible; the other is to limit the number of variables to be monitored. As to the communication overhead, it depends mainly on the time taken for sending the data to the online model checker. If the observer and the online model checker can communicate through a shared memory, then the overhead will be much lower.

Finally, we discuss the influence introduced by online model checking on the system to be checked with respect to the four criteria: functionality, schedulability, reliability, and certifiability. It turns out that our online model checking mechanism does meet these criteria well.

Chapter 7

Case Study

The Traffic Alert and Collision Avoidance System (TCAS) is an on-board aircraft collision detection and resolution system aimed to reduce the incidence of mid-air collisions between aircraft. TCAS is a well-known application in the domain of embedded systems, which has been studied not only in academia but also in industry [100–104]. Therefore, TCAS can be considered as "a benchmark for safety critical applications" [103].

In this chapter, we'd like to demonstrate the applicability of the online model checking technique to a publicly available component of TCAS, which is responsible to provide a solution for the pilot to avoid collision with other aircraft.

7.1 TCAS

TCAS [105] is designed to work independently of the aircraft navigation equipment and the ground systems, which are used to provide Air Traffic Control (ATC) services. An aircraft equipped with TCAS interrogates periodically all other aircraft equipped with a corresponding active transponder in a determined range (i.e., protected volume) about their position. Based on the replies received, TCAS tracks the slant range, altitude, and relative bearing of surrounding traffic. Whenever an *intruder* aircraft is entering the protected volume (as shown in Fig. 7.1 and Fig. 7.2), TCAS issues a *Traffic Advisory* (TA) to assist the pilot in the visual search for the intruder aircraft. TCAS then estimates a time needed to reach the Closest Point of Approach (CPA) with the intruder. This time value is used to calculate the vertical separation between the two aircraft. Depending on the results obtained, TCAS may issue a *Resolution Advisory* (RA) to recommend the pilot that he should either increase or maintain the existing vertical separation from the intruder aircraft.

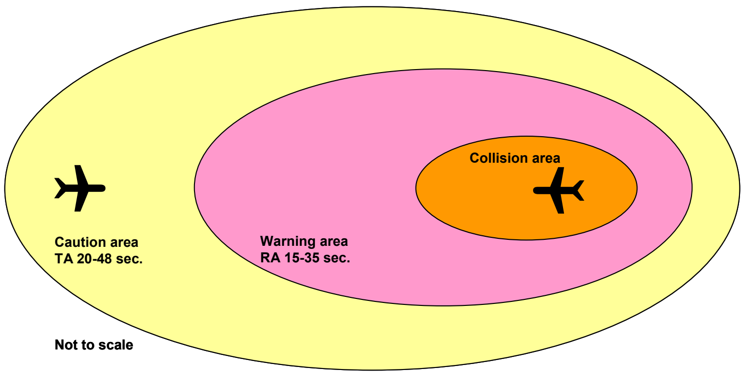

FIGURE 7.1: Protected volume (horizontal view) [106]

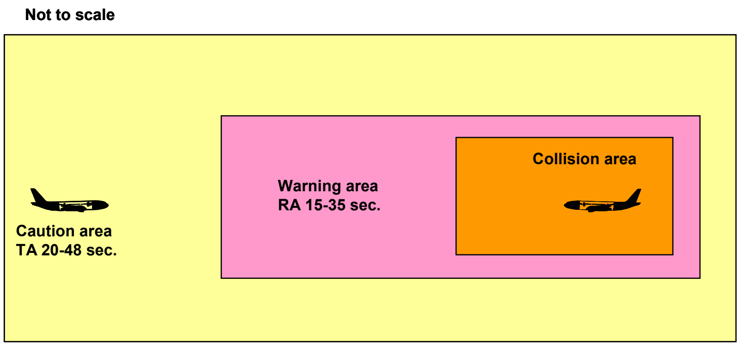

FIGURE 7.2: Protected volume (vertical view) [106]

Hybrid surveillance is a method that decreases the rate of surveillance interrogations made by the TCAS unit of an aircraft [106]: with *active* surveillance, TCAS transmits interrogations to the intruder's transponder, as a reply the transponder provides the information such as range, bearing, and altitude of the intruder; with *passive* surveillance, position data provided by an onboard navigation source (typically based on GPS) is broadcast from the intruder's transponder.

Fig. 7.3 illustrates how the system transitions from passive surveillance to active surveillance as a function of the collision potential [105]: when an intruder is far from being a threat, it is tracked with *passive* surveillance, and the passive surveillance position is validated once per minute with a TCAS active interrogation; when the intruder is a near threat in either altitude or range, but not both, it is tracked with *passive* surveillance, and the passive surveillance position is validated once every 10 seconds with an active TCAS interrogation; when the intruder is a near threat in both altitude and range, it is tracked with *active* surveillance at a 1 Hz interrogation rate, i.e., once per second. The criteria for transitioning from passive to active surveillance is designed to ensure that all TCAS advisories should be based on active surveillance.

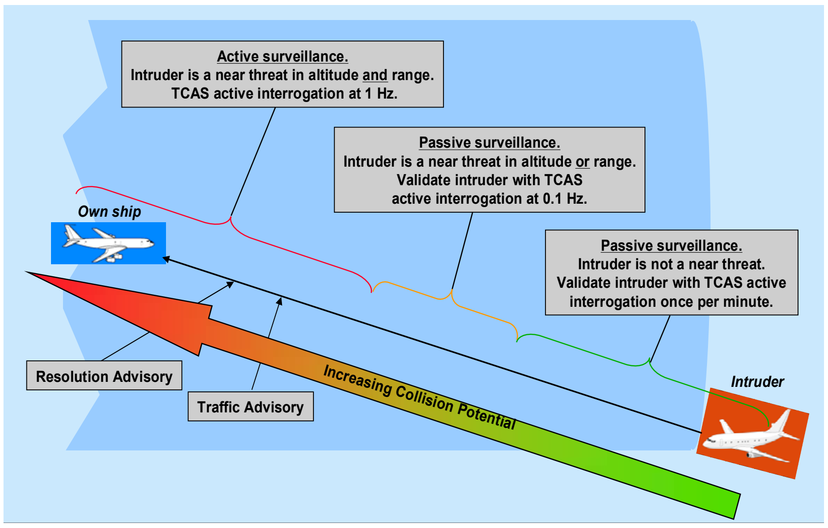
Active surveillance.
Intruder is a near threat in altitude and range.
TCAS active interrogation at 1 Hz.
Passive surveillance.
Intruder is a near threat in altitude or range.
Validate intruder with TCAS
active interrogation at 0.1 Hz.
Passive surveillance.
Intruder is not a near threat.
Validate intruder with TCAS active
interrogation once per minute.
Own ship
Intruder
Resolution Advisory
Traffic Advisory
Increasing Collision Potential

FIGURE 7.3: Transition from passive to active surveillance [105]

7.2 Source Code

In the Software-artifact Infrastructure Repository [107], there is a freely and publicly available RA component of a preliminary version of TCAS in C, called `tacs.c`. The RA component takes as input 12 parameters providing the positions of the two aircraft and returns a single number as its output. The output can be 0, 1, or 2, where 0 means that the situation is unresolved, 1 indicates an upward advisory, and 2 a downward advisory. Based on this output, the aircraft operator is able to decide to increase or decrease the aircraft's altitude. The `main` function of the RA component is given in Algorithm 7.1. From line 159 to line 170 the 12 state variables are set to the current values; line 172 is the special system call for monitoring these variables during runtime. In this case study it is enough to set only one monitoring point. The complete source code of the RA component is provided in Appendix A.1.

ALGORITHM 7.1: The `main` function of the RA component

```
    input: Cur_Vertical_Sep, High_Confidence, Two_of_Three_Reports_Valid, ···
    output: 0,1, or 2
    main(int argc, char *argv[])
    { ······
157   initialize();
158
159   Cur_Vertical_Sep = atoi(argv[1]); //int
160   High_Confidence = atoi(argv[2]); //bool
161   Two_of_Three_Reports_Valid = atoi(argv[3]); //bool
162   Own_Tracked_Alt = atoi(argv[4]); //int
163   Own_Tracked_Alt_Rate = atoi(argv[5]); //int
164   Other_Tracked_Alt = atoi(argv[6]); //int
165   Alt_Layer_Value = atoi(argv[7]); //int
166   Up_Separation = atoi(argv[8]); //int
167   Down_Separation = atoi(argv[9]); //int
168   Other_RAC = atoi(argv[10]); //int
169   Other_Capability = atoi(argv[11]); //int
170   Climb_Inhibit = atoi(argv[12]); //int
171   //special system call for monitoring variables
172   monitor_read(1);
173
174   fprintf(stdout, "%d\n", alt_sep_test());
175   exit(0);
176 }
```

The function call to `alt_sep_test()` at line 174 first tests the minimum vertical separation between two aircraft and then returns an advisory. The definition of this function is given in Algorithm 7.2. First of all, it checks if an upward advisory is needed by calling `Non_Crossing_Biased_Climb()` and `Own_Below_Threat()` (line 124); afterwards, it checks if a downward advisory is needed by calling `Non_Crossing_Biased_Descend()` and `Own_Above_Threat()` (line 125). If neither or both advisories are needed, it returns value 0 (unresolved). Otherwise, it returns the advisory computed.

ALGORITHM 7.2: The definition of the function `alt_sep_test()`

```
    int alt_sep_test()
    { ......
116   enabled = High_Confidence && (Own_Tracked_Alt_Rate <= OLEV) &&
                (Cur_Vertical_Sep > MAXALTDIFF);
117   tcas_equipped = Other_Capability == TCAS_TA;
118   intent_not_known = Two_of_Three_Reports_Valid &&
                         (Other_RAC == NO_INTENT);
119
120   alt_sep = UNRESOLVED;
121
122   if (enabled && ((tcas_equipped && intent_not_known) || !tcas_equipped))
123   {
124     need_upward_RA = Non_Crossing_Biased_Climb() && Own_Below_Threat();
125     need_downward_RA = Non_Crossing_Biased_Descend() &&
                           Own_Above_Threat();
126     if (need_upward_RA && need_downward_RA)
127       /* unreachable: requires both Own_Below_Threat() and
128       Own_Above_Threat() to be true - that requires
129       Own_Tracked_Alt < Other_Tracked_Alt and
130       Other_Tracked_Alt < Own_Tracked_Alt, which isn't possible */
```

```
        alt_sep = UNRESOLVED; //defined as 0
      else if (need_upward_RA)
        alt_sep = UPWARD_RA; //defined as 1
      else if (need_downward_RA)
        alt_sep = DOWNWARD_RA; // defined as 2
      else
        alt_sep = UNRESOLVED;
    }

    return alt_sep;
}
```

7.3 Mapping Functions

As explained in Chapters 4 and 6, we need to map the (concrete) states at the source code level to the corresponding (abstract) state at the model level. For this case study, the 10 mapping functions together with the 12 variables to be monitored are provided in Appendix A.2. Of the 12 variables there are 10 `int` types and 2 `bool` types. The actual values of the 10 `int` variables are passed as parameters to the 10 mapping functions to get 10 `bool` values, as shown in Fig. 7.4. The 10 `bool` values together with the 2 original `bool` values form an abstract state, which will be sent to the online model checker.

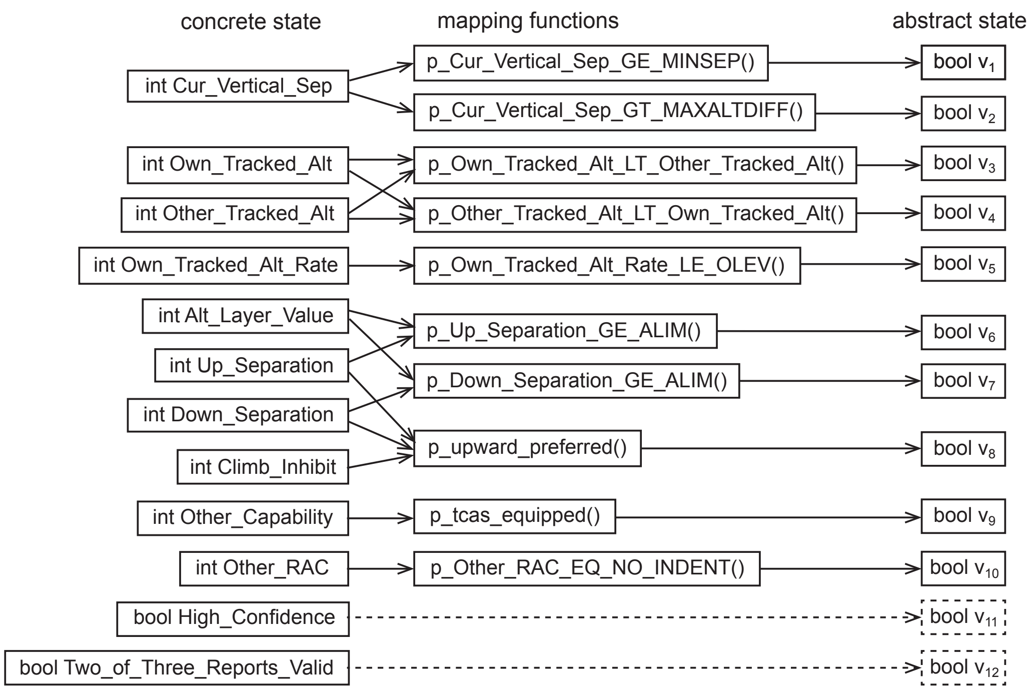

FIGURE 7.4: From concrete state to abstract state

A mapping function[1] usually takes the values of one or more (concrete) state variables as input and returns a `bool` value as output. A typical example is given in Algorithm 7.3. The returned value will be stored into the variable `p_Own_Tracked_Alt_Rate_LE_OLEV`, which is named after the corresponding mapping function. This naming convention can make the mapping relationship more understandable.

ALGORITHM 7.3: Mapping function `p_Own_Tracked_Alt_Rate_LE_OLEV()`

```
input : Own_Tracked_Alt_Rate
output : true or false
bool p_Own_Tracked_Alt_Rate_LE_OLEV()
{
   return (Own_Tracked_Alt_Rate <= OLEV);
}
```

7.4 Abstract Model

In this case study, the mapping functions are deduced from the relational expressions occurring in the function `alt_sep_test()`. By means of replacing the relational expressions with the corresponding `bool` variables, we are able to obtain an *abstract* version of the function `alt_sep_test()` defined in Algorithm 7.4. For example, the relational expression (*Own_Tracked_Alt_Rate* <= `OLEV`) at line 116 in Algorithm 7.2 is replaced by the `bool` variable `p_Own_Tracked_Alt_Rate_LE_OLEV` at line 72 in Algorithm 7.4. Of course, the four functions `Non_Crossing_Biased_Climb()` and `Non_Crossing_Biased_Descend()` as well as `Own_Below_Threat()` and `Own_Above_Threat()` are also redefined in a similar way. The complete definitions are given in Appendix A.3.

ALGORITHM 7.4: The abstract version of the function `alt_sep_test()`

```
   int alt_sep_test()
   { ......
72   enabled = High_Confidence && p_Own_Tracked_Alt_Rate_LE_OLEV &&
               p_Cur_Vertical_Sep_GT_MAXALTDIFF;
73   tcas_equipped = p_tcas_equipped;
74   intent_not_known = Two_of_Three_Reports_Valid &&
                        p_Other_RAC_EQ_NO_INTENT;
75
76   alt_sep = UNRESOLVED;
77
78   if (enabled && ((tcas_equipped && intent_not_known) || !tcas_equipped))
```

[1]In practice, it is better to define each mapping function as *macro* so as to improve the performance of the mapping process.

```
  {
    need_upward_RA = Non_Crossing_Biased_Climb() && Own_Below_Threat();
    need_downward_RA = Non_Crossing_Biased_Descend() &&
                                Own_Above_Threat();
    if (need_upward_RA && need_downward_RA)
      alt_sep = UNRESOLVED; //defined as 0
    else if (need_upward_RA)
      alt_sep = UPWARD_RA; //defined as 1
    else if (need_downward_RA)
      alt_sep = DOWNWARD_RA; // defined as 2
    else
      alt_sep = UNRESOLVED;
  }
  assert(alt_sep != UNRESOLVED);
  return alt_sep;
}
```

In this abstract version of `alt_sep_test`(), the so called abstract state consists of the 12 `bool` variables illustrated in Fig. 7.4. Their values are dependent on the 12 (concrete) variables in the target program and never changed by the program during its execution cycle. If all the 12 variables in the target program are selected to be monitored, then none of the 12 `bool` variables in the behavioral model becomes a free variable. In this case, this abstract version is able to demonstrate the same behavior as the original function. That is, if an error is detected in this abstract version, then there must be an error in the source code of the function `alt_sep_test`().

However, if not all the 12 variables in the target program are monitored, the values of some `bool` variables in the behavioral model may be undefined, i.e., they are free variables. As a consequence, the abstract version is an over-approximation of the original function. This means that an error detected in this abstract version may be spurious, i.e., the checking result is false negative.

7.5 Experimental Results

The experiment is carried out using our prototype implementation on the Linux platform (32 bit Ubuntu 12.04 LTS) with 3GHz Pentium 4 CPU and 2GB RAM (see Section 6.3).

Monitoring State Information There are 12 variables of interest: 10 `int`[2] types and 2 `bool` (`char`) types. Therefore, we need to copy 42 bytes to the `ring buffer`

[2]The size of an `int` is supposed to be 4 bytes on a 32-bit platform.

whenever the monitoring point is reached. The time taken is measured in microseconds for 50 times. On average it takes about $501.78\mu s$ for monitoring the state information of 42 bytes, which is consistent with the result presented in Fig. 6.10 (in Section 6.4.2).

Monitoring Overhead We need to measure the execution time of the task (i.e., the RA component) with and without the system call `monitor_read()` added respectively. The measurement is done 50 times for both cases. The execution time $\mathbf{T}_{task}$ of the task without monitoring state information is about $1319.14\mu s$ on average. Due to monitoring state information, the average execution time of the task is increased by about $\mathbf{T}_{syscall} = 497.7\mu s$. Therefore, the monitoring overhead $\mathbf{P}_{overhead}$ is $\mathbf{T}_{syscall}/\mathbf{T}_{task} = 497.7/1319.14 = 37.7\%$ in this case study.

Sending State Information to Online Model Checker By applying the 10 given mapping functions we obtain 10 `bool` variables. Together with the two original `bool` variables in the source code, we need to send only 12 `bool` (`char`) variables (12 bytes) to the online model checker. The time $\mathbf{T}_{ORCOS\_to\_olmc} = \mathbf{T}_{ORCOS\_to\_QEMU} + \mathbf{T}_{QEMU\_to\_olmc}$ is measured in microseconds and the measurement is also done 50 times. On average we have $\mathbf{T}_{ORCOS\_to\_olmc} = 189.16\mu s$ with $\mathbf{T}_{ORCOS\_to\_QEMU} = 142.3\mu s$ and $\mathbf{T}_{QEMU\_to\_olmc} = 46.86\mu s$ in this case study.

Receiving Checking Result from Online Model Checker As mentioned in Section 6.4.2, since the checking result is only one byte, the transmission time can be seen as constant. We also do the measurement 50 times. On average the transmission time is $\mathbf{T}_{olmc\_to\_ORCOS} = 368.62\mu s$. This result is almost double the time taken for sending the data to the online model checker, because the worker thread (i.e., the delegate) may not always be active at the time when the checking result is sent back to ORCOS.

Online Model Checker The property to be checked is an assertion saying that the function `alt_sep_test()` never returns 0, i.e., `assert(alt_sep != UNRESOLVED)` (line 91 in Algorithm 7.4). By applying the tool CBMC [108] we are able to convert the abstract version of `alt_sep_test()` together with the property into a *boolean* expression in CNF format. In the original program `tacs.c`, the `bool` type is defined as `int` type, which takes 32 bits. For the sake of simplicity, we redefine the `bool` type as `char` type, which takes only 8 bits. The generated CNF file has 2348 (`bool`) variables and 5013 clauses.

Whenever the values of the 12 (abstract) variables are available, the online model checker (with zChaff SAT solver as its search engine) will be invoked to decide whether the CNF formula is satisfiable or not. The measurement is done for 10 randomly generated test cases. The SAT solver takes about $24ms$ on average to finish the checking work.

It is worth mentioning that the generated CNF formula is still highly redundant, because each abstract (state) variable is represented by 8 bits, instead of one bit, in the formula. Therefore, the 12 (abstract) variables need 96 bits, instead of 12 bits, to represent them. The same goes for the other `bool` variables occurring in `alt_sep_test()`. That is, there's still much room to reduce the checking time.

Recall that whenever an intruder becomes a near threat, the rate of surveillance interrogations made by TCAS units is at most 1 Hz, i.e., once per second. In this sense, the monitoring overhead and the communication overhead as well as the model checking overhead are acceptable in this case study.

Theoretically speaking, by applying the online model checking mechanism the property is going to be checked whenever a monitoring point is reached. In this case study, the assertion is going to be checked at line 172 in Algorithm 7.1 before the function `alt_sep_test()` is invoked to run. On the other hand, by introducing the online model checking method we are able to check different properties during program execution without interfering with the target program too much, of course, except inserting monitoring points once and for all in advance.

7.6 Summary

In this chapter we take the RA component of TCAS as case study to demonstrate the applicability of the online model checking approach. In this case study, it is enough to insert just one monitoring point into the source code. There are 10 `int` and 2 `bool` variables that need to be monitored during program execution. By applying 10 mapping functions to the 10 monitored `int` values we are able to obtain 10 `bool` values, which together with the 2 original `bool` values are delivered to the online model checker. The abstract (behavioral) model together with the property to be checked is converted into a *boolean* expression in CNF format. The monitoring overhead and the communication overhead as well as the model checking time are measured using our prototype implementation on the 32 bit Linux platform with 3GHz Pentium 4 CPU and 2GB RAM. The experimental results indicate that the overhead introduced by online model checking is acceptable in this case study. It would be much more efficient to implement our online model checking mechanism including optimization (for this case study) on top of a suitable real hardware platform.

Chapter 8

Online Model Checking for Hybrid Systems

As mentioned in Section 5.1, the embedded applications are a kind of software designed to monitor and control physical processes, which usually results in feedback loops. The software systems are usually modeled by finite state machines, whereas the physical systems are governed in general by differential equations. The former are discrete state systems, while the latter are continuous state systems. Hybrid systems are the combination of the two different worlds. The safety requirements on the continuous dynamics of such systems give rise to a new challenge. In this chapter, we'd like to tackle this challenge by the online model checking mechanism.

8.1 Motivation

Hybrid systems arise in many aspects of our daily life [109], such as aerospace, transportation systems, robotics, motion control, power electronics, and so on. The behaviors of a hybrid system are characterized by the interaction of operating modes and control laws. Each operating mode is associated with a control law in terms of partial or ordinary differential equations or difference equations [110]. The modes are switched following a discrete logic, i.e., a kind of finite state machine.

In hybrid systems, a control law models the behaviors (and disturbance if any) of the physical plant under control. At a mathematical level of abstraction, the control engineer derives from the behaviors of the plants the corresponding control laws as well as the operating modes, which are then optimized and validated by means of analysis and simulation [72]. It is worth mentioning that hybrid systems are often operating in safety-critical situations, such as embedded controllers used in the automotive and airplane industries, and medical devices for monitoring serious health conditions. They are subjected to specific potential failures. Although simulation is an easy way to validate a hybrid system under investigation, it checks only a single trajectory of the system at a time. No matter how many individual trajectories have been checked by simulation, some unsafe case in deep corner may still be missed.

Typical properties studied for hybrid systems are reachability, stability and equilibria, to name only a few. From the perspective of computer science, more attention is paid to the reachability analysis. A hybrid system is considered *safe* if the unsafe states defined in terms of state constraints are not reachable from the initial (safe) states.

For a hybrid system involving continuous dynamics, it is usually difficult to compute and represent the set of states reachable from some initial set [111]. Decidability holds only for those systems with simple continuous dynamics, even the most efficient algorithms for hybrid-system verification usually have exponential time complexity with respect to the dimension of the state space [112]. Recent research [112–116] aims to falsify, instead of verify, the safety of the hybrid systems under investigation, i.e., tries to search for a witness trajectory from an initial state to an unsafe state in case that such a trajectory exists.

In this chapter we focus on ensuring the safety of the continuous dynamics of a hybrid system by means of online model checking. The goal is to falsify the target hybrid system during runtime.

8.2 Hybrid Automaton

Hybrid automata are a kind of formal language for modeling and analyzing the computations consisting of both continuous and discrete dynamics. A hybrid system is usually modeled as a hybrid automaton $H = (Q, X, Inv, E, G, J, U, f, I, F)$ [117], where

- Q is the discrete and finite set of (operating) modes;
- X maps each mode $q \in Q$ to the continuous state space $X_q \subset \mathbf{R}^{dim(X_q)}$;
- Inv maps each mode $q \in Q$ to the continuous invariant $Inv_q \subseteq X_q$;
- $E \subseteq Q \times Q$ is the set of (discrete) transitions between modes;
- G maps each transition $(q_i, q_j) \in E$ to the guard condition $G_{(q_i,q_j)} \subseteq X_{q_i}$;
- J maps each transition $(q_i, q_j) \in E$ to the reset function $J_{(q_i,q_j)} : G_{(q_i,q_j)} \to X_{q_j}$;

- U maps each mode $q \in Q$ to the set of input controls $U_q \subseteq \mathbf{R}^{dim(U_q)}$;
- f maps each mode $q \in Q$ to the continuous dynamics $\dot{x} = f_q(x, u)$ with $u \in U_q$;
- $I \subset Q \times X$ is the set of initial states; and
- $F \subset Q \times X$ is the set of unsafe states.

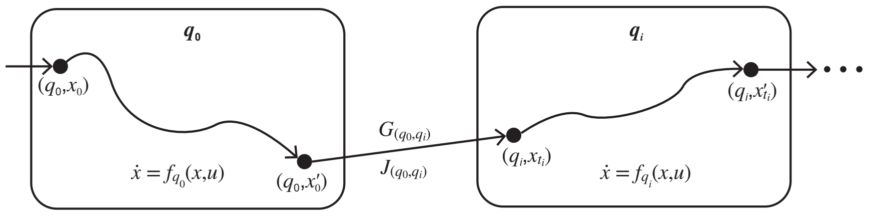

FIGURE 8.1: A hybrid system trajectory

Intuitively, a trajectory of a hybrid system consists of continuous trajectories interleaved with discrete transitions, as illustrated in Fig. 8.1: starting from an initial state (q_0, x_0), the system evolves continuously in time following the control law $\dot{x} = f_{q_0}(x, u)$ as long as the invariant Inv_{q_0} holds. A discrete transition from q_0 to q_i can be triggered if the guard condition $G_{(q_0,q_i)}$ is satisfied. As a result, the state of the system is reset according to the reset function $J_{(q_0,q_i)}$. Let the state now be (q_i, x_{t_i}). Then, the system evolves continuously in time following the control law $\dot{x} = f_{q_i}(x, u)$ until a discrete transition is triggered. The hybrid system behaves repeatedly in this way.

A hybrid automaton is *blocking* if a trajectory has to leave a mode due to the violation of the invariant associated to the mode, but no discrete transition is enabled. A hybrid automaton is *Zeno* if an infinite number of mode switching within a finite time is allowed.

We restrict ourselves to non-blocking and non-Zeno hybrid automata in this chapter.

8.3 Online Falsification Problem

A hybrid system is considered *unsafe* if an unsafe state is reachable from an initial state. For online falsification, we mean to check whether or not a hybrid system is safe while the system is running by online searching for a trajectory of length up to k time steps from an initial state to an unsafe state in the hybrid automaton of the system, as illustrated in Fig. 8.2, where the symbol $\otimes$ represents the unsafe region.

To make online model checking available, we assume that the actual states of the hybrid system under investigation can be probed periodically during runtime. Without loss of generality, let's monitor the actual state information every $T = kdt$ time units, where dt

is the progression of time in one step. The monitored states are stored in a predefined ring buffer. The online model checker tries to take a state from the buffer every T time units. If there is a state available, the online model checker then goes to search for a trajectory from this state to an unsafe state in the hybrid automaton within the time limit T. To reduce the workload of this online search, we compute offline a backward reachable set from the unsafe states up to n time steps beforehand, thus during runtime the online model checker needs only to search in the near future (i.e., up to k time steps for $0 < k < n$) in the state space starting from each monitored state, as illustrated in Fig. 8.3.

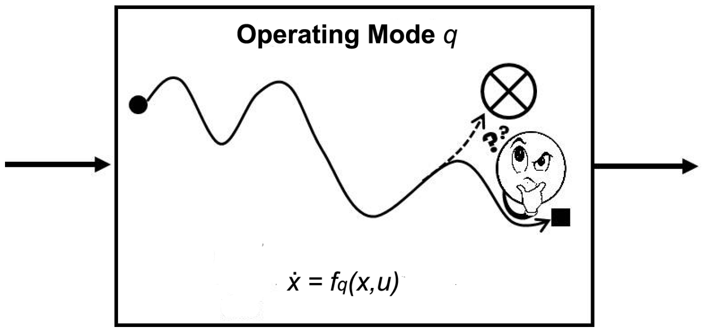

FIGURE 8.2: Online model checking problem

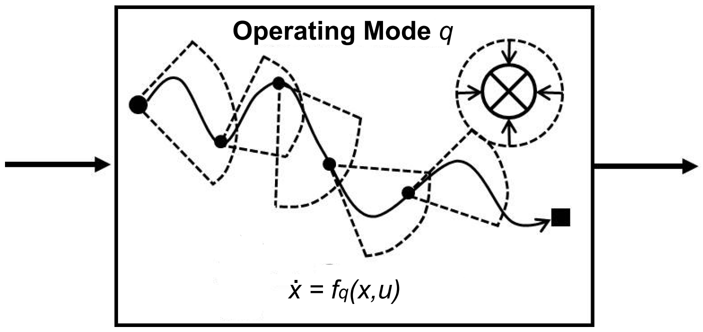

FIGURE 8.3: Online forward reachability checking

In each checking cycle, the online model checker may return the following three possible checking results:

- *unsafe*: the checking process is finished in time, and an error trajectory is found;
- *safe*: the checking process is finished in time, but *no* error trajectory is found;
- *unknown*: the checking process is enforcedly terminated due to timeout.

In the *unsafe* case, where an error trajectory is found within Δ ($\leq T$) time units, ideally the online model checker can predict the error $ndt + kdt - \Delta$ time units in advance. In the *unknown* case, where no error is found within T time units, it is reasonable to believe that there should be no error at least in some neighborhood of the monitored state.

8.4 Offline Backward Reachable Set Computation

Given a hybrid automaton $H = (Q, X, Inv, E, G, J, U, f, I, F)$, the target set F of the unsafe states may include different subset(s) of the continuous state space associated to each (discrete) mode. For a mode $q \in Q$, the set of unsafe states (if any) in X_q can be represented as an implicit surface function $\phi_q : \mathbf{R}^{dim(X_q)} \to \mathbf{R}$ such that $\phi_q(x) \leq 0$ if $(q, x) \in F$ and $\phi_q(x) > 0$ if $(q, x) \notin F$.

The level set methods [118] are a collection of numerical algorithms for computing accurately the evolution of the implicit surface functions following the dynamics defined by Hamilton-Jacobi partial differential equations (PDEs).

The study [119] has proved that "the viscosity solution of a Hamilton-Jacobi-Isaacs (HJI) PDE describes the continuous backward reachable set" and implemented the basic level set methods in MATLAB[1] to calculate the backward reachable set for hybrid systems.

For a mode $q \in Q$ with an unsafe subset in X_q, according to [119], the backward reachable set within X_q is $G_1 \cup G_2 \cup G_3 \setminus E_1$ as illustrated in Fig. 8.4, where

- $G_1 \subset F$ is the initial unsafe subset;
- G_2 is the set of states that can reach the unsafe set(s) in neighbor mode(s) due to uncontrollable input, i.e., disturbance from the environment or the actions of other systems;
- G_3 is the set of states backward reachable from the unsafe set $G_1 \cup G_2$;
- E_1 is the subset of unsafe states in G_3 that can reach the safe state(s) in neighbor mode(s) due to controllable input.

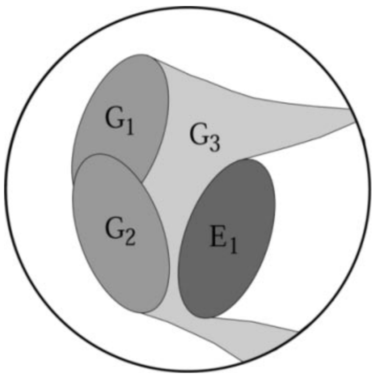

FIGURE 8.4: Computing backward reachable set [120]

In this chapter, we assume that the extended target set, denoted as F^*, has already been offline calculated by the level set methods or any other efficient methods.

[1] www.mathworks.com

8.5 Online Forward Reachability Checking

Given a hybrid automaton H with an extended target set F^*, we need to decide whether or not F^* is reachable within the predefined k time steps starting from each actual state monitored during runtime (see Fig. 8.3). As a proof of concept, we've implemented an online reachability checker based on the iSAT solver.

iSAT [121] is a SATisfiability checker for *Boolean* combinations of arithmetic constraints over real- and integer-valued variables, which can handle not only linear constraints but also non-linear constraints involving transcendental functions. Thanks to a tight integration of DPLL-style SAT solving with interval-based arithmetic constraint propagation, the iSAT solver is able to deal with large *Boolean* combinations of multiple thousand arithmetic constraints over some thousands of variables.

Given a formula to be checked, iSAT solver may terminate in the following three cases:

- ***unsatisfiable***: the formula is actually unsatisfiable;
- ***satisfiable***: the formula is satisfiable[2] and a solution may be generated;
- ***unknown***: a candidate solution may be given, but no guarantee for correctness.

The iSAT solver can conduct bounded reachability checking for hybrid systems, but it can not be directly used for online reachability checking. In cooperation with Karsten Scheibler, a developer of the iSAT solver from the University of Freiburg, an interface for online reachability checking has been developed. Here we introduce mainly the following two functions:

- `isat3_register_trans_and_target(is3, trans, target, k)`: this function unrolls the transition relation `trans` (of the hybrid system model) up to `k` time steps, combines it with the set `target` of the unsafe states, and then registers the generated formula with the iSAT solver `is3`;
- `isat3_solve_with_init(is3, init, T)`: this function reads an actual state `init` monitored during system execution, and then tries to determine at runtime whether or not there exists a trajectory from `init` to `target` within the predefined `T` time units. The learned knowledge is shared by the subsequent calls to this function so as to reduce the computation time.

[2]However, iSAT solver is not able to give a definite answer for some formulas being checked, because interval arithmetic combined with splitting intervals leads to an incomplete deduction calculus.

Let the actual states monitored during system execution be stored in a *buffer*. The online reachability checking algorithm is straightforward as shown in Algorithm 8.1 below.

Algorithm 8.1: Online Bounded Reachability Checking

```
input: $trans$, $target$, $k$, $T$
output: $safe$, $unsafe$, $unknown$
begin
  $formula = isat3\_register\_trans\_and\_target(is3, trans, target, k)$
  $formula = isat3\_node\_simplify\_detroy(is3, formula)$ //simplify the formula
  while ($buffer \neq \emptyset$) do
    $init \longleftarrow buffer$ //read a new actual state
    $result = isat3\_solve\_with\_init(is3, init, T)$
    if ($result == satisfiable$) then output $unsafe$
    elseif ($result == unsatisfiable$) then return $safe$
    else output $unknown$
    endif
  endwhile
end
```

Recall that F^* is obtained by calculating the reachable set from F backward up to n time steps. It is easy to reason that the larger the n, the smaller the k, thus the shorter the online checking time, and the higher the sampling rate for the actual states. It is up to the user to set a proper value to k. Ideally, it is better to make n be a larger value so that errors if any can be predicted in time in most cases.

It is worth pointing out that there exist other efficient tools that can perform forward reachability checking. However, they need to be tailored for *online* reachability checking. We adopt the iSAT solver mainly because it is convenient for us to cooperate with the developers of the iSAT project so as to tailor the tool for our needs. The implementation details of the iSAT tool is beyond the scope of this thesis.

8.6 Experimental Results

Case Study

The RailCab Project [122] (founded at the University of Paderborn in 1997) aims to develop a novel on-demand traffic system for the mobility in the future, whereby small, driverless vehicles (called RailCabs) are able to transport on demand passengers and goods directly to their destination. The RailCabs are equipped with steerable wheels. They can build convoy automatically [123, 124], i.e., driving within small distances without mechanical coupling, so as to reduce the air resistance and the power consumption. A test track in a reduced scale of 1 : 2.5 was built at the University of Paderborn in

2003. Two RailCabs can operate simultaneously at a maximum speed of $10m/s$. The motor can provide the vehicle with acceleration of $\pm 0.8m/s^2$ on planar tracks.

Needless to say, driving in a convoy manner is a safety critical operation. In order to demonstrate the applicability of online model checking to hybrid systems, we design *intentionally* a simplified case study as illustrated in Fig. 8.5: two vehicles RailCab1 and RailCab2 brake along a straight line. The initial distance between the two vehicles is $d_0 = 2.22m$. RailCab1 brakes at the speed $v_1 = 10m/s$ with the constant deceleration $a_1 = -0.8m/s^2$, while RailCab2 brakes at the speed $v_2 = 9.87m/s$ with $a_2 = [-0.8, -0.7]m/s^2$.

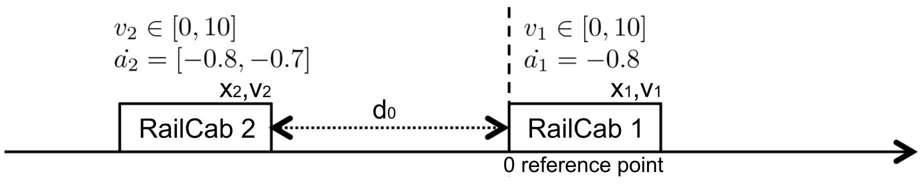

FIGURE 8.5: Case study

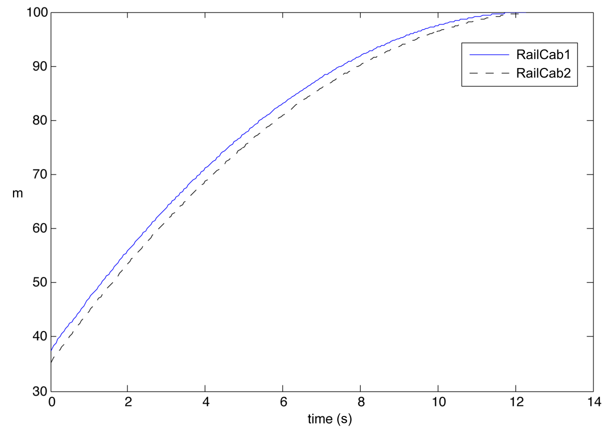

FIGURE 8.6: Distance change over time between the two vehicles

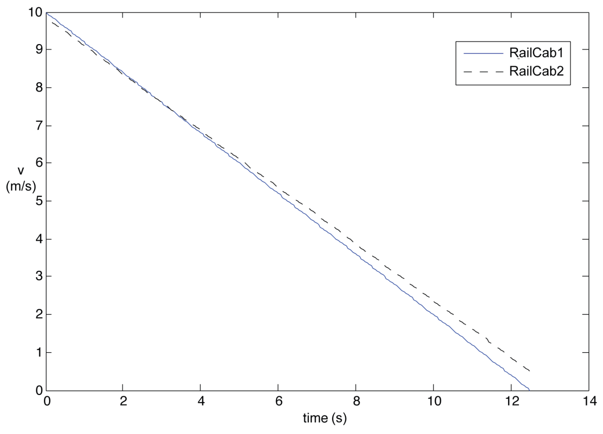

FIGURE 8.7: Speed change over time of the two vehicles

Fig. 8.6 shows the distance change over time between the two vehicles. Fig. 8.7 shows the speed change over time of the two vehicles. That is, the two vehicles should collide with $d = 0m$, $v_1 = 0m/s$ and $v_2 = 0.5m/s$. In other words, $(d = 0, v_1 = 0, v_2 = 0.5)$ is an unsafe state.

Offline Backward Reachable Set

In cooperation with Kathrin Flaßkamp from the Department of Mathematics at University of Paderborn, we calculate the reachable set `target` starting from the unsafe state $(d = 0, v_1 = 0, v_2 = 0.5)$ backward up to $n = 20$ time steps with each (integration) step being $dt = 0.05s$. Since the continuous dynamics of the two vehicles is simple, instead of using the level set methods, we get the backward reachable set using a usual numerical integration algorithm. The numerical solution is illustrated in Fig. 8.8 as a set of points, from each of which there exists a trajectory to the unsafe state $(d = 0, v_1 = 0, v_2 = 0.5)$.

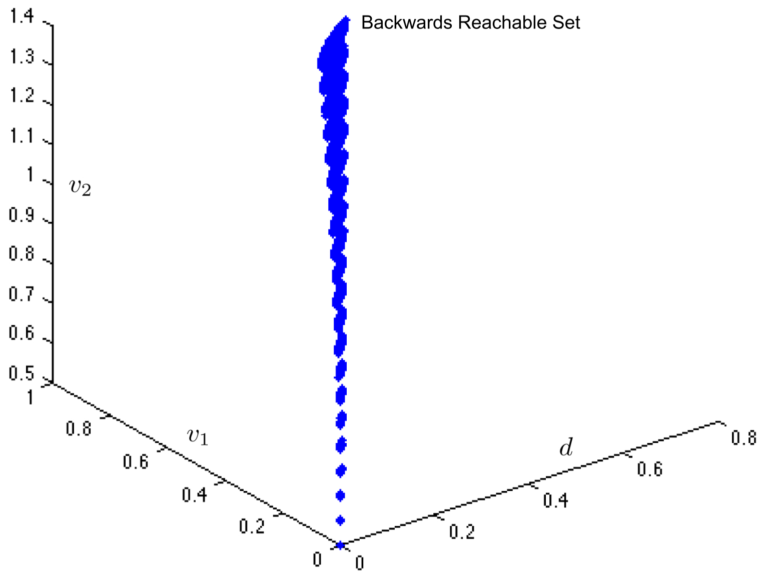

FIGURE 8.8: Backward reachable set

By applying the MATLAB program `vert2lcon` to the backward reachable set `target` in terms of points in Fig. 8.8, we obtain a polyhedron (i.e., a convex hull) illustrated in Fig. 8.9 that covers exactly the points in the backward reachable set. The convex hull determines an implicit surface function, denoted as $\phi : R^3 \rightarrow R$, such that $\phi(x) \leq 0$ if the point $x \in$ `target`; otherwise, $\phi(x) > 0$. The convex hull is defined by the conjunction of 42 linear inequalities of the form $c_1 \cdot d + c_2 \cdot v_1 + c_3 \cdot v_2 \leq c_4$, where c_1, c_2, c_3 and c_4 are constants, as shown in the following list:

$$0.000000d + 1.000000v_1 + -0.000000v_2 \leq 0.800000$$
$$0.694524d + 0.243083v_1 + -0.677161v_2 \leq -0.305591$$
$$0.716920d + 0.215076v_1 + -0.663151v_2 \leq -0.300927$$
$$\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots$$
$$-0.893641d + 0.443749v_1 + 0.067023v_2 \leq 0.033512$$
$$-0.879463d + 0.475460v_1 + 0.021987v_2 \leq 0.010993$$
$$0.000000d + 0.658505v_1 + -0.752577v_2 \leq -0.376288$$

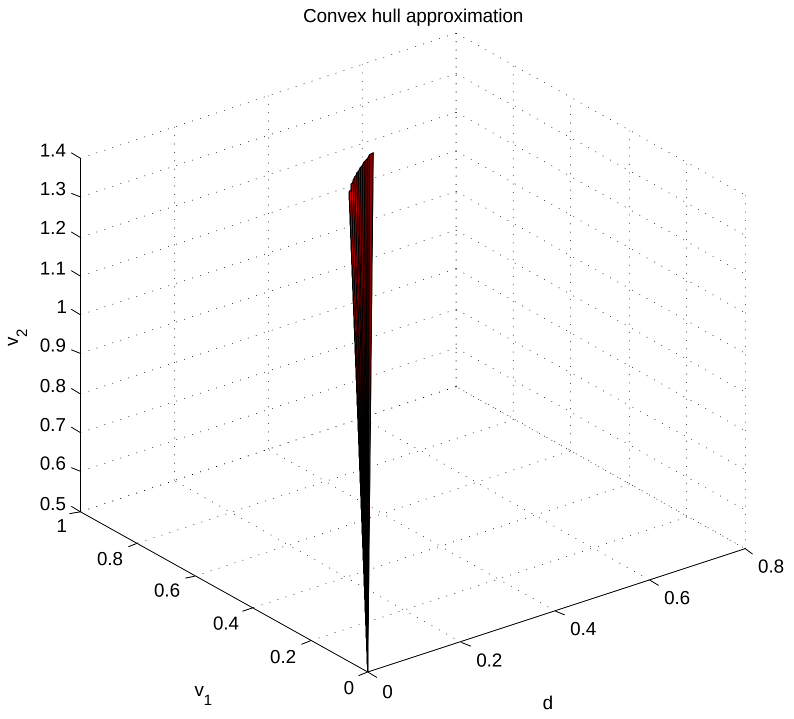

FIGURE 8.9: Convex hull of the backward reachable set

Online Forward Reachability Checking

The transition relation `trans` of the hybrid system model is encoded as a conjunction of the following five expressions:
$x'_1 = x_1 + v_1 dt + 0.5 a_1 dt^2$
$(v'_1 = v_1 + a_1 dt)$ *or* $(v'_1 = 0.0$ *and* $v_1 + a_1 dt \leq 0.0)$ *or* $(v'_1 = v_{max}$ *and* $v_1 + a_1 dt \geq v_{max})$
$x'_2 = x_2 + v_2 dt + 0.5 a_2 dt^2$
$(v'_2 = v_2 + a_2 dt)$ *or* $(v'_2 = 0.0$ *and* $v_2 + a_2 dt \leq 0.0)$ *or* $(v'_2 = v_{max}$ *and* $v_2 + a_2 dt \geq v_{max})$
$t' = t + dt$
where x_1 and x_2 are the positions of the two vehicles relative to a predefined reference point; $dt = 0.05s$ is a constant indicating that the progression of time in one step is dt

seconds. In addition, the primed version of each variable in the above formulas represents the value of the same variable in the next (time) step.

The transition relation describes the behaviors of the two vehicles in one step from the current state (x_1, v_1, x_2, v_2, t) to the next state $(x_1', v_1', x_2', v_2', t')$ with two constraints: (i) if $v_i + a_i dt \leq 0.0$ for $i = 1$ *or* 2, then v_i' remains zero; (ii) if $v_i + a_i dt \geq v_{max}$, then v_i' remains $v_{max} = 10m/s$.

The reachability problem in this case study is encoded as a `formula` by unwinding the `trans` up to $k = 10$ time steps and combining it with the `target` (line 2 in Algorithm 8.1).

Initially, we set $x_1 = 37.5m$, $v_1 = 10.0m/s$, $x_2 = 35.27575m$, $v_2 = 9.87m/s$ and $t = 0.0s$. The current state information $(d = (x_1 - x_2), v_1, v_2)$ is monitored every $0.5s$ by simulating the movement of the two vehicles using a MATLAB program. The generated trajectory is illustrated in Fig. 8.10.

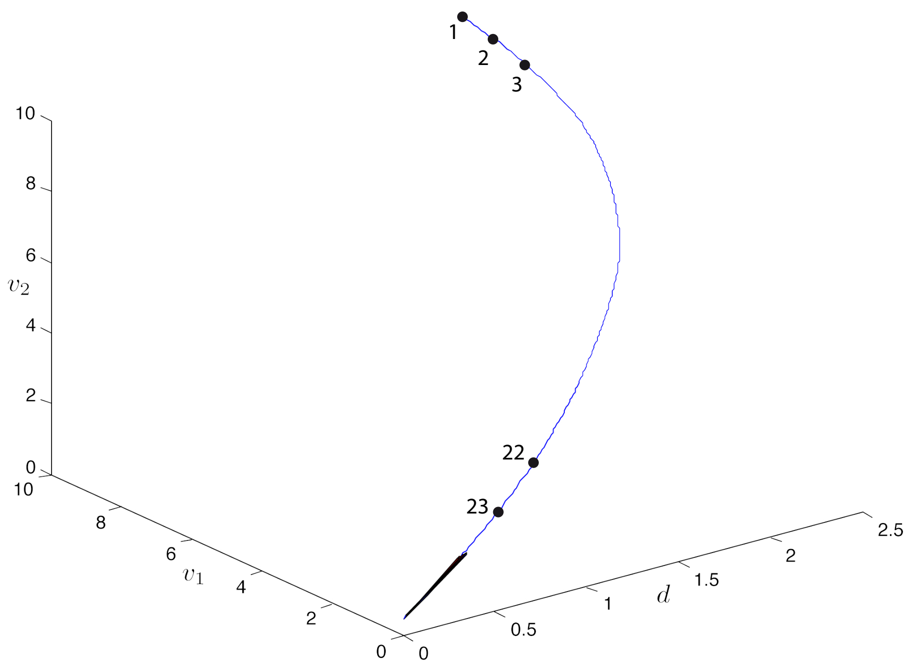

FIGURE 8.10: Online forward reachability checking

We need to check the `formula` during runtime to see whether or not the unsafe `target` is reachable from some monitored state within $k = 10$ time steps (i.e., $kdt = 0.5s$). This is conducted on a Linux platform with 3GHz Intel Core 2 Duo CPU and 4GB RAM. The experimental results are listed in Table 8.1.

As for the former 22 monitored states, the checking results are all *unsatisfiable*, indicating that the unsafe `target` can not be reached within $500ms$. Thus, the online model checker outputs *safe*. The actual time taken for each checking round is not more than $45ms$. As for the 23rd monitored state ($d = 0.69, v_1 = 1.20, v_2 = 1.62$), the iSAT solver does find a solution, indicating that the unsafe `target` may be reached within $500ms$. The online model checker outputs *unsafe* in this case. The actual time taken for this checking round is $110.990ms$.

TABLE 8.1: Experimental results of online forward reachability checking

	Time (ms)	Variables	Clauses	Decisions	Deductions	Conflicts	Result
1	28.815	12358	9852	0	13056	11	*safe*
2	28.049	12412	9895	0	12715	11	*safe*
3	26.832	12458	9938	0	12745	11	*safe*
4	27.086	12500	9981	0	12833	11	*safe*
5	26.695	12535	10019	0	12883	11	*safe*
6	27.509	12565	10052	0	12901	11	*safe*
7	26.968	12613	10095	0	12958	11	*safe*
8	26.915	12659	10138	0	12947	11	*safe*
9	26.729	12703	10181	0	12907	11	*safe*
10	26.584	12794	10224	0	12769	11	*safe*
11	27.168	12777	10257	0	12673	11	*safe*
12	28.015	12819	10300	0	13165	11	*safe*
13	28.219	12974	10343	0	13200	11	*safe*
14	29.283	13177	10386	0	13578	11	*safe*
15	31.379	13541	10429	0	14572	11	*safe*
16	32.612	13883	10462	0	15206	11	*safe*
17	36.017	14101	10505	0	16590	11	*safe*
18	37.314	14089	10548	0	17032	11	*safe*
19	40.483	14030	10591	0	18228	11	*safe*
20	42.964	13597	10629	0	19144	11	*safe*
21	38.913	13307	10662	0	18597	11	*safe*
22	31.174	13009	10705	0	15679	11	*safe*
23	**110.990**	**35289**	**10764**	**28604**	**31814**	**26**	***unsafe***

Recall that we set $n = 20$ and $k = 10$ steps with each step $dt = 50ms$ in this case study. In the 23rd checking round the error is detected within $\varDelta = 111ms$. In ideal case, the online model checker is able to predict the error $ndt + kdt - \varDelta = 1,389ms$ in advance.

8.7 Related Work

The work in [125–127] is also concerned with online model checking for the time-bounded behaviors of a hybrid system in the short-run future. The basic idea is to sample during runtime the numeric values of the observable system state parameters periodically, e.g., a set u of input controls (see Section 8.2). Let the period be T time units. The behavioral model of the hybrid system can thus be reduced by regarding the monitored parameters as *constants* for the next T time units. Obviously, the resulting model is only valid within T time units. As a consequence, the online model checker has to provide an answer within T time units whether or not the reduced model may violate the (safety) properties being checked. To do this, a path-oriented reachability analysis [116] is applied, whereby a feasibility problem of a set of linear constraints is derived and solved by the linear programming (LP) approach.

It is worth pointing out that the state variables of the the reduced behavioral model are *not* monitored at all while doing model checking during runtime. In this sense, this online checking method is also called scenario based verification [125].

E.g., for a communication based train control (CBTC) system under investigation, the radio block center sends the control parameter, i.e., the movement authority[3] (MA), to the onboard system of the trains nearby in every $T = 500ms$. Upon receiving the MA, the onboard computer of each train needs to calculate a legal operation speed range taking into account of the current speed of the train, the status of the track, the wind speed, and others. Notice that the MA keeps unchanged within the next $500ms$. Thus, the behavioral model of the CBTC system can be reduced by regarding the MA as constant. The resulting model is valid within the next $500ms$. Hence, the online model checker needs to provide an answer within $500ms$ whether or not the reduced model may violate the safety property being checked. In this example, the MA is the only variable monitored whenever it is generated or updated. The other state variables in the reduced model are not monitored at all while doing model checking during runtime.

Our online model checking method is different from this work mainly in two aspects as illustrated in Fig. 8.3.

On the one hand, we do not reduce the behavioral model by regarding the monitored parameters as constants and then apply model checking to the reduced model. Instead, we simply sample the state variables of interest in every k (time) steps with each step being dt time units. The online model checker then tries to answer within $T = kdt$ time units whether or not the unsafe target state is reachable within k steps from each monitored state. In our case, the monitored states are supposed to be unchanged within dt time units (i.e., one integration step), instead of T time units. Here the parameters k and dt are determined by the user.

On the other hand, we calculate offline a n-step backward reachable set F^* to reduce the workload of the online model checker.

[3]The distance that a train is authorized to move forward within T time units.

8.8 Conclusion

In this chapter, we extend the application of the online model checking mechanism to hybrid systems. Our goal is to falsify, rather than verify, the safety of the hybrid system under investigation by online reachability analysis. For this purpose, we search for a witness trajectory from each monitored state to an unsafe state in the hybrid system model during system execution. Analogous to the case of discrete state systems, we also calculate offline a n-step backward reachable set from the unsafe states to speed up the reachability checking process during runtime. Of course, parallel computing can also be used to accelerate the reachability checking for hybrid systems. But this is not the focus of this chapter.

As a proof of concept, we've implemented an online reachability checker for hybrid systems using the tailored iSAT solver as its search engine. A simplified case study based on two vehicles operating in a convoy manner demonstrates the applicability of online model checking to hybrid systems. The experimental results indicate that in theory the online model checker is able to predict errors before the errors actually occur.

Chapter 9

Conclusion and Future Work

Nowadays our world depends more and more on embedded systems. They are widely used in industry and are reshaping the way we live. Many of us use embedded systems every day without even knowing it. In this thesis, we concern ourselves with such kind of embedded systems that are safety-critical, whose failure or malfunction may result in severe damages, including loss of life.

9.1 Conclusion

Modern embedded systems are a kind of special-purpose computer systems, which are becoming more complicated due to the advances in electronic techniques. They may fail due to *external* reasons, such as mechanical stress and faulty input, or due to *internal* reasons, such as design errors and physical faults. An increasing number of computer system failures are caused by design errors in software [2]. New software technology tends to enhance the "intelligence" of modern embedded systems. This increases the complexity of embedded software and makes subtle errors extremely difficult to figure out. In practice, no single checking technique, such as testing, simulation, model checking, monitoring, etc., or any combination thereof, is able to completely ensure that the embedded software does behave as desired after it is released or deployed. Against this background, we present our online model checking mechanism as a complementary method.

By doing model checking during system execution, we are able to monitor the actual state information so as to reduce the state space to be explored by the online model checker. The state space explosion problem is thus avoided to some degree by making the online model checker look ahead in each checking cycle only finitely many steps in the state space of the behavioral model of the system under investigation. Online model checking is a lightweight and incomplete method that can falsify, rather than verify, the behavioral model of the target system. The goal is to ensure the correctness of the actual execution trace, instead of the universal correctness, of the target system during runtime (with respect to the property to be checked).

For this purpose, the actual state information is monitored periodically during system execution. Starting from each monitored state in the given behavioral model, the online model checker attempts to find an error path of bounded length within an allocated time limit. The property to be checked is specified in LTL. A nontrivial LTL formula is either safety or liveness or a conjunction of the two. It is sufficient to make the online model checker solve the safety checking problem as well as the liveness checking problem. To this end, we reduce the problem of safety checking and liveness checking to the corresponding invariant checking problem, which can be solved by reachability analysis. As a consequence, our online model checking is in effect a kind of online reachability checking. Because of checking on the model level, the online model checker is able to predict potential errors during runtime.

Reachability checking is also a challenge for large complex systems, not to mention doing it during runtime, which suffers from the limited time allocated to it. We speed up the online reachability checking process by reducing the workload and adopting the symbolic state-based search algorithm as well as using parallel computing. The workload of the online model checker can be reduced by calculating offline a m-step transition relation R^m and a n-step backward reachable set F^* from the target set of error states. According to our experience, making two symbolic state-based model checkers work in parallel can obtain a better price/performance ratio.

We need to instrument the source code of the target system with a finite set of monitoring points once and for all in advance. During system execution, once a monitoring point is reached, the state information at this point will be recorded in a (ring) buffer. In each checking cycle, the online model checker tries to take a state from the buffer and then conducts reachability analysis starting from this state. The monitoring points are determined by analyzing the control flow graph of the target program. We present a partitioning algorithm to calculate a smaller set of monitoring points, which are distributed more or less evenly in the control flow graph.

We present a general framework for integration of online model checking with a real-time operating system, such as ORCOS. This integration framework can be implemented on different hardware architectures from single-core, or multi-core to multiprocessor. We implement a prototype on top of a (virtualized) multicore platform. At each monitoring point, a special system call is introduced together with its system call handler to record the actual state information. In this way, both the source code of the target program and the underlying operating system only need minor modifications. We analyze qualitatively the additional overhead introduced by online model checking. According to our experience, the monitoring overhead can be reduced by setting the native system calls in the source code as monitoring points as long as possible or by limiting the number of variables to be monitored; the communication overhead between the target program and the online model checker depends largely on the underlying system architecture.

We take the RA component of TCAS as case study to demonstrate the applicability of our online model checking method. In this case study, only one monitoring point is inserted into the source code. 10 `int` and 2 `bool` variables are monitored during program execution. By applying 10 mapping functions to the 10 `int` values monitored, we get 10 `bool` values, which together with the original 2 `bool` values are sent to the online model checker. The property to be checked is an assertion. The experimental results indicate that the overhead introduced by online model checking is acceptable in this case study.

We extend the application of our online model checking mechanism to hybrid systems. Analogous to the case of discrete state systems, our goal is to falsify, rather than verify, the safety of the hybrid system under investigation by the online reachability analysis. We search for a witness trajectory of length up to k time steps from each monitored state to an unsafe state in the hybrid system model during system execution. To speed up this reachability checking process, we also calculate offline a n-step backward reachable set from the unsafe states. A simplified case study based on two vehicles operating in a convoy manner demonstrates the applicability of online model checking to hybrid systems. The experimental results indicate that in theory the online model checker is able to predict errors before the errors actually occur.

Online model checking has the following advantages over offline checking techniques:

- Avoid the state space explosion problem to some degree: in each checking cycle, the online model checker tries to search for an error path of bounded length in the state space;
- Detect errors ultra deep in the state space: theoretically, the online model checker is able to reach those states that locate arbitrarily deep in the state space;
- Predict errors before they actually happen: due to checking on the model level, the online model checker is able to predict errors, even if it falls behind the execution of the target system.

Compared with online monitoring, online model checking has the following advantages:

- The behavioral model bridges the semantic gap between the requirements and the source code of the target program;
- Monitoring points are placed in fixed locations in the source code independent of the properties to be checked;
- The distance of any two adjacent monitoring points is not more than k steps;
- Look ahead up to k steps on the model level from each monitored state;
- Liveness properties can be checked on the model level during runtime.

When applying the online model checking mechanism to an embedded software system, the schedulability analysis of the target system can be conducted offline beforehand.

The above mentioned advantages indicate that the online model checking method is complementary to, but can not be replaced by, the existing checking techniques.

9.2 Future Work

Before the online model checking mechanism can be applied to real world applications, there are several basic issues that remain to be investigated:

First, given a monitored state in the behavioral model M, how to calculate efficiently the exact starting points in the state space of $M \times B_{\neg f}$ for the new checking cycle is a topic worth further research.

Second, many programming languages allow dynamic memory allocation and function recursion, which result in dynamic data structures and dynamic function-calling chains respectively. How to monitor the variables stored in the memory allocated dynamically during runtime as well as how to determine the monitoring points in the source code written in such programming languages is worth further study.

Third, whenever an error is detected by the online model checker, how to evaluate the severity of the error is a topic worth further consideration.

Last but not least, once the detected error is identified as a severe error, how to deal with this error during runtime is also worth further investigation.

Appendix A

Case Study: TCAS

A.1 RA Component of TCAS

```c
/*  -*- Last-Edit:  Fri Jan 29 11:13:27 1993 by Tarak S. Goradia; -*- */
/* $Log: tcas.c,v $
 * Revision 1.2  1993/03/12  19:29:50  foster
 * Correct logic bug which didn't allow output of 2 - hf
 * */

#include <stdio.h>

#define OLEV        600      /* in feets/minute */
#define MAXALTDIFF  600      /* max altitude difference in feet */
#define MINSEP      300      /* min separation in feet */
#define NOZCROSS    100      /* in feet */

typedef int bool;

/* variables */
int Cur_Vertical_Sep;
bool High_Confidence;
bool Two_of_Three_Reports_Valid;

int Own_Tracked_Alt;
int Own_Tracked_Alt_Rate;
int Other_Tracked_Alt;

int Alt_Layer_Value;         /* 0, 1, 2, 3 */
int Positive_RA_Alt_Thresh[4];

int Up_Separation;
int Down_Separation;

int Other_RAC;          /* NO_INTENT, DO_NOT_CLIMB, DO_NOT_DESCEND */
#define NO_INTENT 0
#define DO_NOT_CLIMB 1
#define DO_NOT_DESCEND 2

int Other_Capability;       /* TCAS_TA, OTHER */
#define TCAS_TA 1
#define OTHER 2

int Climb_Inhibit;      /* true/false */

#define UNRESOLVED 0
#define UPWARD_RA 1
#define DOWNWARD_RA 2

void initialize()
{
    Positive_RA_Alt_Thresh[0] = 400;
    Positive_RA_Alt_Thresh[1] = 500;
    Positive_RA_Alt_Thresh[2] = 640;
    Positive_RA_Alt_Thresh[3] = 740;
}

int ALIM ()
{
    return Positive_RA_Alt_Thresh[Alt_Layer_Value];
}

int Inhibit_Biased_Climb ()
{
    return (Climb_Inhibit ? Up_Separation + NOZCROSS : Up_Separation);
}

bool Non_Crossing_Biased_Climb()
{
    int upward_preferred;
    int upward_crossing_situation;
    bool result;

    upward_preferred = Inhibit_Biased_Climb() > Down_Separation;
    if (upward_preferred)
    {
        result = !(Own_Below_Threat()) || ((Own_Below_Threat()) &&
                        (!(Down_Separation >= ALIM())));
    }
    else
    {
        result = Own_Above_Threat() && (Cur_Vertical_Sep >= MINSEP) &&
                             (Up_Separation >= ALIM());
    }
    return result;
}

bool Non_Crossing_Biased_Descend()
{
    int upward_preferred;
    int upward_crossing_situation;
    bool result;

    upward_preferred = Inhibit_Biased_Climb() > Down_Separation;
    if (upward_preferred)
    {
        result = Own_Below_Threat() && (Cur_Vertical_Sep >= MINSEP) &&
                             (Down_Separation >= ALIM());
    }
    else
    {
        result = !(Own_Above_Threat()) || ((Own_Above_Threat()) &&
                             (Up_Separation >= ALIM()));
    }
    return result;
}

bool Own_Below_Threat()
{
    return (Own_Tracked_Alt < Other_Tracked_Alt);
}

bool Own_Above_Threat()
{
    return (Other_Tracked_Alt < Own_Tracked_Alt);
}

int alt_sep_test()
{
    bool enabled, tcas_equipped, intent_not_known;
    bool need_upward_RA, need_downward_RA;
    int alt_sep;

    enabled = High_Confidence && (Own_Tracked_Alt_Rate <= OLEV) &&
                             (Cur_Vertical_Sep > MAXALTDIFF);
    tcas_equipped = Other_Capability == TCAS_TA;
    intent_not_known = Two_of_Three_Reports_Valid && (Other_RAC ==
                             NO_INTENT);

    alt_sep = UNRESOLVED;

    if (enabled && ((tcas_equipped && intent_not_known) || !tcas_equipped))
    {
        need_upward_RA = Non_Crossing_Biased_Climb() && Own_Below_Threat();
        need_downward_RA = Non_Crossing_Biased_Descend() &&
                    Own_Above_Threat();
        if (need_upward_RA && need_downward_RA)
            /* unreachable: requires Own_Below_Threat and Own_Above_Threat
             to both be true - that requires
             Own_Tracked_Alt < Other_Tracked_Alt and
             Other_Tracked_Alt < Own_Tracked_Alt, which isn't possible */
            alt_sep = UNRESOLVED;
        else if (need_upward_RA)
            alt_sep = UPWARD_RA;
        else if (need_downward_RA)
            alt_sep = DOWNWARD_RA;
        else
            alt_sep = UNRESOLVED;
    }

    return alt_sep;
}

main(argc, argv)
int argc;
char *argv[];
{
    if(argc < 13)
    {
        fprintf(stdout, "Error: Command line arguments are\n");
        fprintf(stdout, "Cur_Vertical_Sep, High_Confidence,
                    Two_of_Three_Reports_Valid\n");
        fprintf(stdout, "Own_Tracked_Alt, Own_Tracked_Alt_Rate,
                    Other_Tracked_Alt\n");
        fprintf(stdout, "Alt_Layer_Value, Up_Separation,
                    Down_Separation\n");
        fprintf(stdout, "Other_RAC, Other_Capability, Climb_Inhibit\n");
        exit(1);
    }

    initialize();

    Cur_Vertical_Sep = atoi(argv[1]);
    High_Confidence = atoi(argv[2]);
    Two_of_Three_Reports_Valid = atoi(argv[3]);
    Own_Tracked_Alt = atoi(argv[4]);
    Own_Tracked_Alt_Rate = atoi(argv[5]);
    Other_Tracked_Alt = atoi(argv[6]);
    Alt_Layer_Value = atoi(argv[7]);
    Up_Separation = atoi(argv[8]);
    Down_Separation = atoi(argv[9]);
    Other_RAC = atoi(argv[10]);
    Other_Capability = atoi(argv[11]);
    Climb_Inhibit = atoi(argv[12]);

    monitor_read(1); /*special system call for monitoring variables*/

    fprintf(stdout, "%d\n", alt_sep_test());
    exit(0);
}
```

A.2 Monitored Variables and Mapping Functions

```
#include <stdio.h>

typedef int bool;

/* monitored variables */
int Cur_Vertical_Sep;
bool High_Confidence;
bool Two_of_Three_Reports_Valid;

int Own_Tracked_Alt;
int Own_Tracked_Alt_Rate;
int Other_Tracked_Alt;

int Alt_Layer_Value;        /* 0, 1, 2, 3 */

int Up_Separation;
int Down_Separation;

int Other_RAC;          /* NO_INTENT, DO_NOT_CLIMB, DO_NOT_DESCEND */
int Other_Capability;       /* TCAS_TA, OTHER */
int Climb_Inhibit;      /* true/false */

#define OLEV        600       /* in feets/minute */
#define MAXALTDIFF  600       /* max altitude difference in feet */
#define MINSEP      300           /* min separation in feet */
#define NOZCROSS    100       /* in feet */

#define NO_INTENT 0
#define DO_NOT_CLIMB 1
#define DO_NOT_DESCEND 2
```

```

#define TCAS_TA 1
#define OTHER 2

#define UNRESOLVED 0
#define UPWARD_RA 1
#define DOWNWARD_RA 2

#define Positive_RA_Alt_Thresh_0 = 400
#define Positive_RA_Alt_Thresh_1 = 500
#define Positive_RA_Alt_Thresh_2 = 640
#define Positive_RA_Alt_Thresh_3 = 740

/* mapping functions */
bool p_Cur_Vertical_Sep_GE_MINSEP()
{
    return (Cur_Vertical_Sep >= MINSEP);
}

bool p_Own_Tracked_Alt_LE_Other_Tracked_Alt()
{
    return (Own_Tracked_Alt < Other_Tracked_Alt);
}

bool p_Other_Tracked_Alt_LE_Own_Tracked_Alt()
{
    return (Other_Tracked_Alt < Own_Tracked_Alt);
}

bool p_Down_Separation_GE_ALIM()
{
    int Positive_RA_Alt_Thresh;

    switch (Alt_Layer_Value)
    {
        case 0:
            Positive_RA_Alt_Thresh = Positive_RA_Alt_Thresh_0;
            break;
        case 1:
            Positive_RA_Alt_Thresh = Positive_RA_Alt_Thresh_1;
            break;
        case 2:
            Positive_RA_Alt_Thresh = Positive_RA_Alt_Thresh_2;
            break;
        case 3:
            Positive_RA_Alt_Thresh = Positive_RA_Alt_Thresh_3;
            break;
        default:
            break;
    }

    return (Down_Separation >= Positive_RA_Alt_Thresh);
}

bool p_Up_Separation_GE_ALIM()
{
    int Positive_RA_Alt_Thresh;

    switch (Alt_Layer_Value)
    {
        case 0:
            Positive_RA_Alt_Thresh = Positive_RA_Alt_Thresh_0;
            break;
        case 1:
            Positive_RA_Alt_Thresh = Positive_RA_Alt_Thresh_1;
            break;
        case 2:
            Positive_RA_Alt_Thresh = Positive_RA_Alt_Thresh_2;
            break;
        case 3:
            Positive_RA_Alt_Thresh = Positive_RA_Alt_Thresh_3;
            break;
        default:
            break;
    }
    return (Up_Separation >= Positive_RA_Alt_Thresh);
}

bool p_upward_preferred()
{
    return ((Climb_Inhibit ? Up_Separation + MINSEP : Up_Separation) >
                      Down_Separation);
}

bool p_tcas_equipped()
{
    return (Other_Capability == TCAS_TA);
}

bool p_Own_Tracked_Alt_Rate_LE_OLEV()
{
    return (Own_Tracked_Alt_Rate <= OLEV);
}

bool p_Cur_Vertical_Sep_GT_MAXALTDIFF()
{
    return (Cur_Vertical_Sep > MAXALTDIFF);
}

bool p_Other_RAC_EQ_NO_INTENT()
{
    return (Other_RAC == NO_INTENT);
}
```

A.3 Abstract Model

```
typedef int bool;

/*abstract variables*/
bool p_Cur_Vertical_Sep_GE_MINSEP;
bool p_Cur_Vertical_Sep_GT_MAXALTDIFF;
bool p_Own_Tracked_Alt_LT_Other_Tracked_Alt;
bool p_Other_Tracked_Alt_LT_Own_Tracked_Alt;
bool p_Own_Tracked_Alt_Rate_LE_OLEV;
bool p_Up_Separation_GE_ALIM;
bool p_Down_Separation_GE_ALIM;
bool p_upward_preferred;
bool p_tcas_equipped;
bool p_Other_RAC_EQ_NO_INTENT;

bool High_Confidence;
bool Two_of_Three_Reports_Valid;

#define UNRESOLVED 0
#define UPWARD_RA 1
#define DOWNWARD_RA 2

bool Own_Below_Threat()
{
    return p_Own_Tracked_Alt_LT_Other_Tracked_Alt;
}

bool Own_Above_Threat()
{
    return p_Other_Tracked_Alt_LT_Own_Tracked_Alt;
}

bool Non_Crossing_Biased_Climb()
{
    bool upward_preferred;
    bool result;

    upward_preferred = p_upward_preferred;
    if (upward_preferred)
    {
        result = !(Own_Below_Threat()) || ((Own_Below_Threat()) &&
                               !(p_Down_Separation_GE_ALIM));
    }
    else
    {
        result = Own_Above_Threat() && p_Cur_Vertical_Sep_GE_MINSEP &&
                               p_Up_Separation_GE_ALIM;
    }
    return result;
}

bool Non_Crossing_Biased_Descend()
{
    bool upward_preferred;
    bool result;

    upward_preferred = p_upward_preferred;
    if (upward_preferred)
    {
        result = Own_Below_Threat() && p_Cur_Vertical_Sep_GE_MINSEP &&
                               p_Down_Separation_GE_ALIM;
    }
    else
    {
        result = !(Own_Above_Threat()) || ((Own_Above_Threat()) &&
                               p_Up_Separation_GE_ALIM);
    }
    return result;
}

int alt_sep_test()
{
    bool enabled, tcas_equipped, intent_not_known;
    bool need_upward_RA, need_downward_RA;
    int alt_sep;

    enabled = High_Confidence && p_Own_Tracked_Alt_Rate_LE_OLEV &&
                               p_Cur_Vertical_Sep_GT_MAXALTDIFF;
    tcas_equipped = p_tcas_equipped;
    intent_not_known = Two_of_Three_Reports_Valid &&
                               p_Other_RAC_EQ_NO_INTENT;

    alt_sep = UNRESOLVED;

    if (enabled && ((tcas_equipped && intent_not_known) || !tcas_equipped))
    {
        need_upward_RA = Non_Crossing_Biased_Climb() && Own_Below_Threat();
        need_downward_RA = Non_Crossing_Biased_Descend() &&
                              Own_Above_Threat();
        if (need_upward_RA && need_downward_RA)
            alt_sep = UNRESOLVED;
        else if (need_upward_RA)
            alt_sep = UPWARD_RA;
        else if (need_downward_RA)
            alt_sep = DOWNWARD_RA;
        else
            alt_sep = UNRESOLVED;
    }
    assert(alt_sep!=UNRESOLVED);
    return alt_sep;
}
```

List of Figures

List of Tables

List of Selected Publications

1. Krishna Sudhakar, Yuhong Zhao, and Franz-Josef Rammig. Efficient Integration of Online Model Checking into a Small Footprint Real-Time Operating System. In *Proc. 17th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, pages 374-383, June 2014.

2. Franz Rammig, Lial Khaluf, Norma Montealegre, Katharina Stahl, and Yuhong Zhao. Organic Real-Time Programming – Vision and Approaches towards Self-evolving and Adaptive Real-time Software. In Lynn Choi and Raimund Kirner, editors, *Proc. 9th Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS'13).* IEEE, June 2013.

3. Mona Qanadilo, Sufyan Samara, and Yuhong Zhao. Accelerating Online Model Checking. In *Proceedings of the 6th Latin-American Symposium on Dependable Computing (LADC'13)*, pages 40-47, April 2013.

4. Yuhong Zhao and Franz Rammig. Online Model Checking for Dependable Real-Time Systems. In *Proc. 15th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, pages 154-161, Shenzhen, China, April 2012. ISBN 978-1-4673-0499-3.

5. Sufyan Samara, Yuhong Zhao, and Franz-Josef Rammig. Integrate Online Model Checking into Distributed Reconfigurable System on Chip with Adaptable OS Services. In *Distributed, Parallel and Biologically Inspired Systems.* volume 329 of IFIP Advances in Information and Communication Technology. pages 102-113. Springer Boston, September 2010.

6. Franz-Josef Rammig, Yuhong Zhao, and Sufyan Samara. Online Model Checking as Operating System Service. In *Proc. 7th IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS'09)*. IFIP WG 10.5, Springer, November 2009.

7. Yuhong Zhao, Simon Oberthür, and Franz-Josef Rammig. Runtime Model Checking for Safety and Consistency of Self-Optimizing Mechatronic Systems. In *Proceedings of the 7th International Heinz Nixdorf Symposium: Self-Optimzing Mechatronic Systems*. ALB-HNI-Verlagsschriftenreihe. Heinz Nixdorf Institut, February 2008.

8. Yuhong Zhao, Martin Kardos, Simon Oberthür, and Franz-Josef Rammig. Comprehensive Verification Framework for Dependability of Self-Optimizing Systems. In *Proceedings of the 3rd International Symposium on Automated Technology for Verification and Analysis (ATVA'05)*. Taipei, Taiwan, October 2005.

9. Yuhong Zhao, Simon Oberthür, Martin Kardos, and Franz-Josef Rammig. Model-based Runtime Verification Framework for Self-Optimizing Systems. In *Proceedings of the 2005 Workshop on Runtime Verification (RV'05)*. Edinburgh, Scotland, UK, July 2005.

Bibliography

[1] Christof Ebert and Capers Jones. Embedded software: Facts, figures, and future. *Computer*, 42(4):42–52, April 2009. ISSN 0018-9162.

[2] Hermann Kopetz. *Real-time systems: design principles for distributed embedded applications*. Kluwer international series in engineering and computer science: Real-time systems. Kluwer Academic Publishers, 2011. ISBN 9780792398943.

[3] Donna Tam. Google's Sergey Brin: You'll ride in robot cars within 5 years. *CNet News*, September 25. 2012.

[4] Jaikumar Vijayan. Amazon's efforts to test drones for package delivery gain support. *Computerworld News*, Aug 27. 2014.

[5] Walt Truszkowski, Harold Hallock, Christopher Rouff, Jay Karlin, James Rash, Michael Hinchey, and Roy Sterritt. *Autonomous and Autonomic Systems: With Applications to NASA Intelligent Spacecraft Operations and Exploration Systems*. Springer, 2009. ISBN 1846282322.

[6] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, January 2003. ISSN 0018-9162.

[7] Philip Koopman, Charles Shelton, Beth Latronico, and Jen Morris. Roses: Robust self-configuring embedded systems. URL http://www.ece.cmu.edu/~koopman/roses/.

[8] Collaborative Research Centre 614. Self-optimizing concepts and structures in mechanical engineering. URL http://www.sfb614.de/en/home/.

[9] Franz-Josef Rammig, Lial Khaluf, Norma Montealegre, Katharina Stahl, and Yuhong Zhao. Organic real-time programming —vision and approaches towards self-evolving and adaptive real-time software. In Lynn Choi and Raimund Kirner, editors, *Proc. 9th Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS'13)*. IEEE, 17 - 18 2013.

[10] Arjan van der Schaft and Hans Schumacher. *An introduction to hybrid dynamical systems*, volume 251 of *Lecture Notes in Control and Information Sciences*. Springer, London, 2000.

[11] C.W. Johnson and C.M. Holloway. The dangers of failure masking in fault-tolerant software: Aspects of a recent in-flight upset event. In *Proceedings of the 2nd Institution of Engineering and Technology International Conference on System Safety*, pages 60–65, Oct 2007.

[12] Alwyn Goodloe and Lee Pike. Monitoring distributed real-time systems: A survey and future directions. Technical Report NASA/CR-2010-216724, NASA Langley Research Center, July 2010.

[13] David M. Cummings. Haven't found that software glitch, Toyota? Keep trying. *Los Angeles Times*, March 11. 2010.

[14] Associated Press. Safety agency studying Toyota acceleration problem. *Fox News*, September 30. 2014.

[15] Jim Turley. Cars, Coding, and Carelessness. *Electronic Engineering Journal*, November 27. 2013.

[16] Andrew Glover. In pursuit of code quality: Monitoring cyclomatic complexity. *IBM developerWorks*, March 28. 2006.

[17] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004. ISBN 0471469122.

[18] Edmund M. Clarke, Orna Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[19] J. van Leeuwen, J. Hartmanis, and G. Goos, editors. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[20] Sergey Berezin, Sérgio Vale Aguiar Campos, and Edmund M. Clarke. Compositional Reasoning in Model Checking. In *COMPOS'97: Revised Lectures from the International Symposium on Compositionality: The Significant Difference*, pages 81–102, London, UK, 1998. Springer-Verlag. ISBN 3-540-65493-3.

[21] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994. ISSN 0164-0925.

[22] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:118–149, 2003.

[23] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification, 12th International Conference (CAV 2000)*, pages 154–169. Springer, 2000. ISBN 3-540-67770-4.

[24] Daniel Kroening and Ofer Strichman. Efficient computation of recurrence diameters. In *Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI 2003, pages 298–309, London, UK, UK, 2003. Springer-Verlag. ISBN 3-540-00348-7.

[25] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, January 2004. ISSN 1545-5971.

[26] Felix Salfner, Maren Lenk, and Miroslaw Malek. A survey of online failure prediction methods. *ACM Comput. Surv.*, 42(3):10:1–10:42, 2010. ISSN 0360-0300.

[27] A. Avizienis. The four-universe information system model for the study of fault tolerance. *Proceedings of the 12th Annual International Symposium on Fault-Tolerant Computing, Santa Monica, California*, pages pp. 6–13, 1982.

[28] B. Plattner and J. Nievergelt. Special feature: Monitoring program execution: A survey. *Computer*, 14(11):76–93, Nov 1981. ISSN 0018-9162.

[29] Amir Pnueli. The temporal semantics of concurrent programs. *Theor. Comput. Sci.*, 13:45–60, 1981.

[30] Bowen Alpern and Fred B. Schneider. Recognizing Safety and Liveness. *Distributed Comp.*, 2:117–126, 1987.

[31] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, March 1977. ISSN 0098-5589.

[32] Marco Daniele, Fausto Giunchiglia, and Moshe Y. Vardi. Improved automata generation for linear temporal logic. In *CAV*, pages 249–260, 1999.

[33] Fabio Somenzi and Roderick Bloem. Efficient büchi automata from ltl formulae. In *CAV*, pages 248–263, 2000.

[34] Paul Gastin and Denis Oddoux. Fast ltl to büchi automata translation. In *CAV*, pages 53–65, 2001.

[35] Xavier Thirioux. Simple and efficient translation from ltl formulas to büchi automata. *Electr. Notes Theor. Comput. Sci.*, 66(2), 2002.

[36] Dimitra Giannakopoulou and Flavio Lerda. From states to transitions: Improving translation of ltl formulae to büchi automata. In *FORTE*, pages 308–326, 2002.

[37] Carsten Fritz. Constructing büchi automata from linear temporal logic using simulation relations for alternating büchi automata. In *CIAA*, pages 35–48, 2003.

[38] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. A system of specification patterns. URL `http://patterns.projects.cis.ksu.edu`.

[39] Pierre Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1/2):72–99, January/February 1983.

[40] Orna Kupferman and Moshe Y. Vardi. Model checking of safety properties. *Form. Methods Syst. Des.*, 19(3):291–314, October 2001. ISSN 0925-9856.

[41] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992. ISSN 0360-0300.

[42] Nina Amla, Robert P. Kurshan, Kenneth L. McMillan, and Ricardo Medel. Experimental analysis of different techniques for bounded model checking. In Hubert Garavel and John Hatcliff, editors, *TACAS*, volume 2619 of *Lecture Notes in Computer Science*, pages 34–48. Springer, 2003. ISBN 3-540-00898-5.

[43] Kenneth L. McMillan and Nina Amla. Automatic abstraction without counterexamples. In *Proceedings of the 9th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'03, pages 2–17, Berlin, Heidelberg, 2003. Springer. ISBN 3-540-00898-5.

[44] N. Delgado, A.Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *Software Engineering, IEEE Transactions on*, 30 (12):859–872, Dec 2004. ISSN 0098-5589.

[45] Clinton Jeffery, Wenyi Zhou, Kevin Templer, and Michael Brazell. A lightweight architecture for program execution monitoring. In *Proceedings of the ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 67–74, New York,USA, 1998. ACM. ISBN 1-58113-055-4.

[46] Sriram Sankar and Manas Mandal. Concurrent runtime monitoring of formally specified programs. *Computer*, 26(3):32–41, March 1993. ISSN 0018-9162.

[47] Bernd Bruegge, Tim Gottschalk, and Bin Luo. A framework for dynamic program analyzers. In *Proceedings of the 8th annual conference on Object-oriented programming systems, languages, and applications*, OOPSLA'93, pages 65–82, New York, USA, 1993. ACM. ISBN 0-89791-587-9.

[48] Michel Diaz, Guy Juanole, and Jean-Pierre Courtiat. Observer-a concept for formal on-line validation of distributed systems. *IEEE Trans. Softw. Eng.*, 20(12): 900–913, December 1994. ISSN 0098-5589.

[49] A. Q. Gates, S. Roach, O. Mondragon, and N. Delgado. DynaMICs: Comprehensive Support for Run-Time Monitoring. In *Proceedings of the First Workshop on Runtime Verification (RV'01)*, pages 61–77. Elsevier, 2001.

[50] Weiming Gu, Greg Eisenhauer, Karsten Schwan, and Jeffrey Vetter. Falcon: Online monitoring for steering parallel programs. In *In Ninth International Conference on Parallel and Distributed Computing and Systems (PDCS'97)*, pages 699–736, 1998.

[51] Detlef Bartetzko, Clemens Fischer, Michael Möller, and Heike Wehrheim. Jass - java with assertions. *Electr. Notes Theor. Comput. Sci.*, 55(2):103–117, 2001.

[52] Klaus Havelund and Grigore Roşu. Java pathexplorer - a runtime verification tool. In *Proceedings of the 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space: A New Space Odyssey*, June 2001.

[53] Moonjoo Kim, Mahesh Viswanathan, Hanêne Ben-Abdallah, Sampath Kannan, Insup Lee, and Oleg Sokolsky. MAC: A Framework for Run-time Correctness Assurance of Real-Time Systems. Technical Report MS-CIS-98-37, University of Pennsylvania, 1998.

[54] Feng Chen and Grigore Roşu. Towards Monitoring-Oriented Programming: A Paradigm Combining Specification and Implementation. In *Proceedings of the Workshop on Runtime Verification (RV'03)*, volume 89(2) of *ENTCS*, pages 108 – 127, 2003.

[55] Jeffery. J. P. Tsai, K.-Y. Fang, H.-Y. Chen, and Y.-D. Bi. A noninterference monitoring and replay mechanism for real-time software testing and debugging. *IEEE Trans. Softw. Eng.*, 16(8):897–916, August 1990. ISSN 0098-5589.

[56] Sarah Chodrow and Mohamed G. Gouda. Implementation of the sentry system. *In Software Practice and Experience*, 25:373–387, 1994.

[57] Doron Drusinsky. The Temporal Rover and the ATG Rover. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 323–330, London, UK, 2000. Springer. ISBN 3-540-41030-9.

[58] Marta Kwiatkowska. Quantitative verification: Models techniques and tools. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 449–458, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-811-4.

[59] Radu Calinescu, Carlo Ghezzi, Marta Kwiatkowska, and Raffaela Mirandola. Self-adaptive software needs quantitative verification at runtime. *Commun. ACM*, 55 (9):69–77, September 2012. ISSN 0001-0782.

[60] Gerd Behrmann, Kim Guldstrand Larsen, and Radek Pelánek. To store or not to store. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV)*, volume 2725 of *Lecture Notes in Computer Science*, pages 433–445. Springer, 2003.

[61] Yuhong Zhao and Franz Rammig. Online Model Checking for Dependable Real-Time Systems. In *Proceedings of the 15th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, pages 154–161, Shenzhen, China, April 2012. IEEE Computer Society. ISBN 978-1-4673-0499-3.

[62] Viktor Schuppan. *Liveness checking as safety checking to find shortest counterexamples to linear time properties.* PhD thesis, ETH Zurich, 2006.

[63] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 411–420, New York, NY, USA, 1999. ACM. ISBN 1-58113-074-0.

[64] Radek Pelánek. Beem: benchmarks for explicit model checkers. In *Proceedings of the 14th international SPIN conference on Model checking software*, pages 263–267, Berlin, Heidelberg, 2007. Springer. ISBN 978-3-540-73369-0.

[65] S. Baehni, R. Baldoni, R. Guerraoui, and B. Pochon. The driving philosophers. In *Proceedings of the 3rd IFIP International Conference on Theoretical Computer Science (TCS)*, 2004.

[66] Bryon Moyer, editor. *Real World Multicore Embedded Systems.* Newnes, 2013.

[67] Joe 'Zonker' Brockmeier. Containers vs. Hypervisors: Choosing the Best Virtualization Technology. *The Linux Foundation*, April 13. 2010.

[68] Chris Ault. Challenges of safety-critical multi-core systems. *Embedded Newsletter*, April 23. 2011.

[69] Mona Qanadilo, Sufyan Samara, and Yuhong Zhao. Accelerating online model checking. In *Proceedings of the 6'th Latin-American Symposium on Dependable Computing (LADC'13)*, pages 40–47, April 2013.

[70] O. Shacham and E. Zarpas. Tuning the VSIDS Decision Heuristic for Bounded Model Checking. In *Proc. Fourth International Workshop on Microprocessor Test and Verification, Common Challenges and Solutions (MTV 2003)*. IEEE Computer Society, 2003.

[71] Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In *2007 Future of Software Engineering*, FOSE '07, pages 37–54, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2829-5.

[72] Thomas A. Henzinger, Benjamin. Horowitz, and Christoph M. Kirsch. Giotto: a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91 (1):84–99, Jan 2003. ISSN 0018-9219.

[73] Farnam Jahanian and Aloysius K. Mok. Modechart: a specification language for real-time systems. *Software Engineering, IEEE Transactions on*, 20(12):933–947, Dec 1994. ISSN 0098-5589.

[74] Manfred Broy, Sascha Kirstan, Helmut Krcmar, Bernhard Schätz, and Jens Zimmermann. What is the benefit of a model-based design of embedded software systems in the car industry? In Jörg Rech and Christian Bunse, editors, *Emerging Technologies for the Evolution and Maintenance of Software Models*, pages 343–369. IGI, 2011.

[75] Steven P. Miller, Elise A. Anderson, Lucas G. Wagner, Michael W. Whalen, and Mats P. E. Heimdahl. Formal verification of flight critical software. In *Proceedings of AIAA Guidance, Navigation, and Control Conference*, San Francisco, USA, August 2005.

[76] Matthew Staats and Mats Per Erik Heimdahl. Partial translation verification for untrusted code-generators. In Shaoying Liu, T. S. E. Maibaum, and Keijiro Araki, editors, *Proceedings of the10th International Conference on Formal Engineering Methods (ICFEM'08)*, pages 226–237, Kitakyushu, Japan, October 2008. ISBN 978-3-540-88193-3.

[77] MISRA. *Guidelines for the Use of the C Language in Vehicle Based Software.* Motor Industry Research Association, Nuneaton, UK, 1998. ISBN 0-9524159-9-0.

[78] Motor Industry Software Reliability Association. *Guidelines for the Use of the C Language in Critical Systems.* Mira Books Limited, 2004. ISBN 0-9524156-2-3.

[79] MISRA. *Guidelines for the Use of the C Language in Critical Systems.* Motor Industry Research Association, Warwickshire, UK, 2013. ISBN 978-1-906400-10-1.

[80] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, New York, USA, 1992. ISBN 0-387-97664-7.

[81] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4):21:1–21:54, October 2009. ISSN 0360-0300.

[82] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970. ISSN 0362-1340.

[83] Mijung Kim and K. Selçuk Candan. Sbv-cut: Vertex-cut based graph partitioning using structural balance vertices. *Data Knowl. Eng.*, 72:285–303, February 2012. ISSN 0169-023X. With permission of Elsevier.

[84] Krishna Sudhakar. Integrating a real time operating system with an online model checker. Master thesis, Heinz Nixdorf Institute, University of Paderborn, Paderborn, Germany, December 2013.

[85] Krishna Sudhakar, Yuhong Zhao, and Franz-Josef Rammig. Efficient integration of online model checking into a small-footprint real-time operating system. In *Proc. 2014 IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, pages 374–383, June 2014.

[86] Colin Walls. Multicore basics: AMP and SMP. *Embedded Newsletter*, March 15. 2014.

[87] AUTOSAR. Specification of Operating Systems (Version 5.0.0). Technical report, Automotive Open System Architecture GbR, 2011.

[88] Franz-Josef Rammig, Yuhong Zhao, and Sufyan Samara. Online model checking as operating system service. In *Proc. 7th IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS'09)*. IFIP WG 10.5, Springer, November 2009.

[89] John A. Carbone. An RTOS for an SMP Multicore Processor. *Software Tools and Techniques (Special Section)*, October 2006.

[90] M. Vaidehi and T. R. Gopalakrishnan Nair. Multicore applications in real time systems. *CoRR*, abs/1001.3539, 2010. URL http://arxiv.org/abs/1001.3539.

[91] Research Group Rammig at University of Paderborn. ORCOS (Organic ReConfigurable Operating System). URL https://orcos.cs.uni-paderborn.de/orcos/www/.

[92] Franz-Josef Rammig, Katharina Stahl, and Gavin Vaz. A framework for enhancing dependability in self-x systems by artificial immune systems. In Uwe Brinkschulte, M. Theresa Higuera-Toledano, and Achim Rettberg, editors, *Proc. 4th IEEE Workshop on Self-Organizing Real-Time Systems (SORT) 2013*. IEEE, June 2013.

[93] Timo Kerstan and Simon Oberthür. A configurable hybrid kernel for embedded real-time systems. In Achim Rettberg, editor, *Proceedings of the International Embedded Systems Symposium*. IFIP WG 10.5, Springer-Verlag, 29 May - 1 June 2007.

[94] Carsten Ditze. A step towards operating system synthesis. In *Proc. of the 5th Annual Australasian Conf. on Parallel And Real-Time Systems (PART). IFIP, IEEE*, 1998.

[95] Giorgio C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series)*. Springer-Verlag TELOS, Santa Clara, CA, USA, 2004. ISBN 0387231374.

[96] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973. ISSN 0004-5411.

[97] Marco Spuri and Giorgio Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *REAL-TIME SYSTEMS*, 10:179–210, 1996.

[98] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC'05, pages 41–46, Berkeley, CA, USA, 2005. USENIX Association.

[99] Markus Becker. Univ. of Paderborn, Paderborn, Germany. Personal comm., 2015.

[100] Nancy G. Leveson, Mats Per Erik Heimdahl, Holly Hildreth, and Jon Damon Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–706, September 1994.

[101] Mats P. E. Heimdahl. Experiences and lessons from the analysis of TCAS II. In *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA'96, pages 79–83, New York, USA, 1996. ACM. ISBN 0-89791-787-1.

[102] William Chan, Richard J. Anderson, Paul Beame, Steve Burns, Francesmary Modugno, David Notkin, and Jon D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):156–166, 1998.

[103] Alberto Coen-Porisini, Giovanni Denaro, Carlo Ghezzi, and Mauro Pezzé. Using symbolic execution for verifying safety-critical systems. In *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-9, pages 142–151, New York, USA, 2001. ACM. ISBN 1-58113-390-1.

[104] Arnaud Gotlieb. TCAS software verification using constraint programming. *Knowledge Engineering Review*, 27(3):343–360, July 2012. ISSN 0269-8889.

[105] Federal Aviation Administration (FAA) U.S. Department of Transportation. Introduction to TCAS II version 7.1, February 2011.

[106] The European Organization for the Safety of Air Navigation (EUROCONTROL). ACAS II Guide, January 2012.

[107] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*, 10(4):405–435, October 2005. ISSN 1382-3256.

[108] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004. ISBN 3-540-21299-X.

[109] Michael S. Branicky and Sven Erik Mattsson. Simulation of hybrid systems. In *Hybrid Systems IV*, pages 31–56, London, UK, 1997. Springer-Verlag. ISBN 3-540-63358-8.

[110] MichaelS. Branicky. Introduction to hybrid systems. In Dimitrios Hristu-Varsakelis and WilliamS. Levine, editors, *Handbook of Networked and Embedded Control Systems*, Control Engineering, pages 91–116. Birkhäuser Boston, 2005. ISBN 978-0-8176-3239-7.

[111] Claire Tomlin, Ian Mitchell, Alexandre M. Bayen, and Meeko Oishi. Computational techniques for the verification of hybrid systems. In *Proceedings of the IEEE*, 91(7):986–1001, 2003.

[112] Erion Plaku, Lydia E. Kavraki, and Moshe Y. Vardi. Hybrid systems: From verification to falsification by combining motion planning and discrete search. *Formal Methods in System Design*, 34(2):157–182, April 2009. ISSN 0925-9856.

[113] Amit Bhatia and Emilio Frazzoli. Incremental search methods for reachability analysis of continuous and hybrid systems. In *Proceedings of Hybrid Systems: Computation and Control*, pages 142–156. Springer, 2004.

[114] M.S. Branicky, M.M. Curtiss, J. Levine, and S. Morgan. Sampling-based planning, control and verification of hybrid systems. In *IEE Proceedings of Control Theory and Applications*, 153(5):575–590, 2006. ISSN 1350-2379.

[115] Lei Bu, Jianhua Zhao, and Xuandong Li. Path-oriented reachability verification of a class of nonlinear hybrid automata using convex programming. In Gilles Barthe and Manuel Hermenegildo, editors, *Proceedings of Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 5944 of *Lecture Notes in Computer Science*, pages 78–94. Springer, 2010. ISBN 978-3-642-11318-5.

[116] Lei Bu and Xuandong Li. Path-oriented bounded reachability analysis of composed linear hybrid systems. *International Journal on Software Tools for Technology Transfer*, 13(4):307–317, 2011. ISSN 1433-2779.

[117] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, February 1995. ISSN 0304-3975.

[118] Stanley Osher and Ronald Fedkiw. *Level Set Methods and Dynamic Implicit Surfaces*. Springer, November 2002. ISBN 0387954821.

[119] Ian Mitchell. *Application of Level Set Methods to Control and Reachability Problems in Continuous and Hybrid Systems*. PhD thesis, Stanford University, Stanford, USA, August 2002.

[120] Ian M. Mitchell and Claire J. Tomlin. Overapproximating reachable sets by hamilton-jacobi projections. *Journal of Scientific Computing*, 19(1):323–346, 2003. ISSN 1573-7691. With permission of Springer.

[121] Martin Fränzle, Christian Herde, Tino Teige, Stefan Ratschan, and Tobias Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *Journal on Satisfiability, Boolean Modeling and Computation*, 1:209–236, 2007.

[122] University of Paderborn. RailCab Preject. URL `http://www.railcab.de`.

[123] C. Henke, N. Fröhleke, and J. Bcker. Advanced convoy control strategy for autonomously driven railway vehicles. In *IEEE Conf. on Intelligent Transportation Systems (ITSC)*, 2006.

[124] Christian Henke, Matthias Tichy, Tobias Schneider, Joachim Böcker, and Wilhelm Schäfer. System architecture and risk management for autonomous railway convoys. In *Proceedings of the 2nd Annual IEEE International Systems Conference, Montreal, Canada*, April 2008.

[125] Lei Bu, Xin Chen, Linzhang Wang, and Xuandong Li. Online verification of control parameter calculations in communication based train control system. *Computing Research Repository (CoRR)*, abs/1101.4271, 2011.

[126] Lei Bu, Qixin Wang, Xin Chen, Linzhang Wang, Tian Zhang, Jianhua Zhao, and Xuandong Li. Toward online hybrid systems model checking of cyber-physical systems' time-bounded short-run behavior. *SIGBED Rev.*, 8(2):7–10, June 2011. ISSN 1551-3688.

[127] Tao Li, Feng Tan, Qixin Wang, Lei Bu, Jian-Nong Cao, and Xue Liu. From Offline Toward Real-Time: A Hybrid Systems Model Checking and CPS Co-design Approach for Medical Device Plug-and-Play (MDPnP). In *Proceedings of the 2012 IEEE/ACM Third International Conference on Cyber-Physical Systems*, ICCPS'12, pages 13–22, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4695-7.