On the Design and Implementation of Reliable and Economical Telematics Software Architectures for Embedded Systems. A Domain-specific Framework.

A dissertation to obtain the degree of
doctor rerum naturalium
submitted to the
University of Paderborn,
Faculty of Computer Science,
Electrical Engineering and Mathematics
by Dipl.-Wirt.-Inf. Jan Stehr, Kassel, Germany

2nd November 2010

Abstract

The massive growth of mobile telecommunications and computing power of mobile phone devices over the past decade established an extensive infrastructure for distributed services. Of the corresponding applications, *Electronic Toll Collection* (ETC) is a prime example for mission-critical telematics systems. Consequently, ETC may serve as an archetype in the elaboration of an architecture for reliable and economic telematics software, as presented here.

Due to a lack of substantiated publications, this work first establishes a software view on the ETC domain. It differentiates related domains and describes the defining aspects as well as generic concepts, setup and processes.

With the focus narrowed on the distributed, embedded software components, the initial requirements of domain-specific reliability and economy are refined and substantiated. Submitting the previously specified deployment to a systematic Fault Tree Analysis (FTA) produces a scheme for risk evaluation of telematics solutions as well as a first set of detailed architectural requirements. Approaches to optimized utilization of the costly ETC infrastructure lead to the second set of requirements. These resulting, intended software properties are complemented by high-integrity constraints to modeling and implementation of the architecture.

As primary result, this work presents a set of fundamental design patterns, establishing the HIRTE – a *High-Integrity Run-Time Environment* for mission-critical telematics applications, especially ETC. Conforming to the domain-specific requirements and constraints, the patterns represent building blocks of a framework for state-based service components; statically implementable, exhaustively monitor- and controllable. In this context, the *Virtual Control Unit* (VCU) provides an original, lean virtual machine for crucial applications, extended by the notion of the *Distributable State Machine Fragment* (DSMF), enabling selective, safe delegation of processing steps. As a proof of concept as well as foundation for future research and development, all patterns feature a reference implementation in a language subset of Ada.

Application architecture patterns complement the fundamental design patterns, structuring them in a framework context, refining their implementation roles and finally sketching an ETC software smartphone solution based on the introduced original concepts.

Keywords: telematics, telecommunications, smart cards, electronic toll collection, automotive, fault-awareness, high-integrity, mission-critical components, virtual machines, state automatons, Ada.

Zusammenfassung

Das vergangene Jahrzehnt sah ein massives Wachstum sowohl mobiler Telekommunikation als auch der Rechenleistung von Mobiltelefonen. Daraus entstand eine breit verfügbare Infrastruktur für verteilte Dienste. In der Menge der damit verbundenen Anwendungen stellt die *elektronische Mauterhebung* (Electronic Toll Collection; ETC) ein herausragendes Beispiel für hochkritische Telematiksysteme dar. Bei der Erarbeitung einer Architektur für zuverlässige und wirtschaftliche Telematiksoftware, wie hier präsentiert, kann ETC daher als ein Archetyp dienen.

Wegen des Mangels an fundierten Veröffentlichungen führt diese Arbeit zunächst die Domäne ETC aus Sicht der Software ein. Dabei grenzt sie verwandte Gebiete ab und beschreibt die definierenden Aspekte, wie auch generische Konzepte, den Aufbau und Prozesse.

Ein Fokus auf den verteilten, eingebetteten Software-Komponenten verfeinert und konkretisiert die initialen Anforderungen domänenspezifischer Zuverlässigkeit und Wirtschaftlichkeit. Die systematische Analyse der potentiellen Defekte (Fault Tree Analysis; FTA) liefert ein Schema zur Risikoevaluierung von Telematiklösungen, als auch einen ersten Satz detaillierter Anforderungen an die Architektur. Ansätze zur optimierten Auslastung der aufwändigen ETC-Infrastruktur führen zu einem zweiten Satz von Anforderungen. Die so gegebenen angestrebten Eigenschaften der Software werden durch Randbedingungen für hochintegre Modellierung und Implementierung der Architektur ergänzt.

Als primäres Ergebnis präsentiert die vorliegende Arbeit eine Reihe von grundlegenden Entwurfsmustern. Damit führt sie die HIRTE (*High-Integrity Run-Time Environment*) ein – eine hochintegre Laufzeitumgebung für kritische Telematikanwendungen, insbesondere ETC. Konform zu den domänenspezifischen Anforderungen und Randbedingungen bilden die Muster Bausteine eines Realisierungsrahmens für zustandsbasierte Dienstkomponenten; statisch implementierbar, erschöpfend kontrollier- und steuerbar. In diesem Zusammenhang stellt die *virtuelle Steuereinheit* (Virtual Control Unit; VCU) eine neuartige, schlanke virtuelle Maschine für kritische Applikationen zur Verfügung, erweitert durch den Ansatz eines *verteilbaren Zustandsmaschinenfragments* (Distributable State Machine Fragment; DSMF), welches die selektive und sichere Auslagerung von Verarbeitungsschritten ermöglicht. Als Nachweis der Konzepttauglichkeit sowie Grundlage für künftige Forschung und Entwicklung verfügen alle Entwurfsmuster über eine Referenzimplementierung in einer Sprachuntermenge von Ada.

Applikationsarchitekturmuster vervollständigen die grundlegenden Entwurfsmuster, strukturieren sie in einem Framework-Kontext, verfeinern ihre Rollen in der Implementierung und skizzieren zum Abschluss eine ETC-Softwarelösung für Smartphones, basierend auf den vorgestellten neuen Ansätzen.

Stichworte: Telematik, Telekommunikation, Chipkarten, automatische Mauterhebung, Automotive, Mängelbewusstsein, Hochintegrität, kritische Komponenten, virtuelle Maschinen, Zustandsautomaten, Ada.

Contents

List of Figures

List of Tables

Listings

One

Introduction

1.1 Motivation

dawn of mobile infrastructure

The mobile telecommunications sector saw a massive technological development and commercial growth over the last decade. Along with the increasing capabilites of mobile devices, the ubiquity of the internet and decline of data traffic costs, today we find an efficient, readily available infrastructure for distributed services.

telematics in general

One of the hype topics in this context, the term of *telematics* was adopted by many projects. An amalgam of *telecommunications* and *informatics*, we designate it an infrastructure composed of distributed components that interact with each other or server components over some radio net, a fixed line or mobile telecommunications network. Application domains are diverse and range from healthcare (e.g. the German *Elektronische Gesundheitskarte*) over traffic management and renewable energies (smart grids) to defense (e.g. autonomous systems and drones).

just for convenience

Many telematics solutions provide convenience services to a user, i.e. their output generates some benefit, but failure does not result in objective damage (beyond inconvenience). A navigation system represents a proper example found in the domain of traffic telematics: if it works, it smoothly leads the user[1] to a selected destination – if not, a paper map or passers-by may help. Annoying, but no harm done.

... and in earnest

On the other hand, more and more *mission-critical* applications employ telematics infrastructures. These systems realize services which process information that represents or is associated with an artifact of actual and immediate importance: the well-being of a person, integrity of a vehicle or simply a large sum of money. Consequently, failure of such a service may cause substantial damage. Characteristic examples include telemedicine (remotely monitoring vital signs) in healthcare and automatic emergency calls in traffic telematics.

[1]A regular user, not paramedics, firefighers etc.

health versus money

And one more step of differentiation: mission-critical relating to *potential physical harm* of human beings or *monetary losses.* This distinction is fundamental for the avenues available to system development to handle failures. In the former case, efforts in time and money are less an issue, because literally "life is invaluable". Stringent effectiveness, not necessarily efficiency, permits approaches like multiple redundancy – clear, robust, but costly solutions. Merely monetary risks, however, demand subtly different concepts: the system design needs to balance costs to achieve reliability with the potential pecuniary losses. So, now it is about reliability *and* economy.

ETC as representative

A scientifically rewarding subject to further research this specific type of mission-critical telematics system is the domain of *Electronic Toll Collection* (ETC), more precisely *Global Navigation Satellite System/Cellular Networks* (GNSS/CN) solutions:

- An ETC system handles massive amounts of money; billions of Euros per year that require complete and consistent audit trails. Quality of service requirements are unforgiving and very specific. The operator may be held liable for the system overstepping any of the defined fault thresholds. Penalties can easily reach many million Euros a day.
- It features a complex, heterogenous infrastructure with embedded automotive units, interacting with center servers over CNs, with roadside devices over various short-range media – all that reproducibly and securely conforming to elaborate business processes.
- Correspondingly, these business processes call for the seamless and secure integration of a multitude of software components and services, ranging from navigation to payment and communications via different interfaces, all in a distributed automotive environment.
- With a harmonization on a European level politically and legislatively settled, the technical implementation of the ambitious EU concepts and directives will provide a challenge for some time.
- The current rapid development and dissemination of devices like smartphones and services of the mobile internet is very likely to impact the closed, proprietary ETC systems eventually, demanding answers to the questions of safe and reliable integration of open components.

domain-related approaches

When a development team has to tackle a project of the magnitude of ETC, preparations include the selection of domain-specific software engineering approaches to follow. Naturally, available valid and proven methods (and corresponding tools) are candidates for customization and integration into a defined development process. If one surveys the seemingly innumerable initiatives and works on software in the context of telecommunications, telematics, automotive and embedded systems, the somehow related topic of ETC seems well (if not exhaustively) covered. On close examination however, many of the works are not as readily applicable to the domain as the close technological relation might suggest.

… and their shortcomings

First of all, often, the published experts have either a rather hypothetical understanding of these systems, or their practical experience is limited to consulting, pilots and prototypes or – see above – non-critical software (navigation, infotainment, multimedia). Second, due to the variety and complexity of applications, they have to either concentrate on specific details of the development (like modeling, e.g. [Do04]), with disregard to determining dependencies to other steps in the process (e.g. feasibility analysis, validation, coding), or they leave out any details whatsoever and keep a rather orbital distance to their subject (e.g. [HR02]). Third, the ostensible similarity of use cases – e.g. identification of a road by navigation and by road tolling – can prove treacherous, leading to the assumption that similar development approaches must surely be fitting, then. This completely ignores the very different quality of service requirements; the distinction between convenience and mission-criticality (see above).

classic versus software engineering

In contrast to that, the classic engineering sciences are already well established in the domain of ETC: works e.g. on signal processing and communications in a motorway environment are available, as are exhaustive documents on road pricing-specific positioning, locating and dead reckoning. Corresponding dedicated works on software engineering and architecture however, are quite rare – but still software plays a crucial role in these deployments, determining functionality and processes, shaping the service chain, and contributing significantly to the overall quality of service.

reliability of an ETC system

When a government authority commissions an ETC operator, or a toll charger an EETS provider (cmp. [EU09] or 5.3), it has basically two (non-exclusive) options to safeguard its interests: 1) hold the operator/provider financially liable for any disruption, thus ensuring steady revenue by means of a contract, or 2) exact a solid proof of reliability. Both options offer challenges to software engineering. In the case that something goes awry – i.e. loss of revenue –, the operator has to consistently prove down to a technical level that it cannot be held liable, or at least that it is able to systematically identify the problem and its solution quickly to prevent further damage and minimize losses. Proving the reliability means that the operator and its suppliers have to systematically ensure that the probability of revenue loss is acceptably low.

about software architecture

This is a characteristic example for the specific properties of mission-critical telematics systems handling material values. With the discovered deficiency of software engineering and architecture works on this topic, narrowed down to the software components, the distributed elements – in the case of ETC the on-board equipment (OBE) applications – are the most rewarding subjects for research:

- They are immediately associated with the primary value creation of the system, thus realizing crucial and irreplaceable elements of the business process.

- They operate in a restrictive, embedded environment, adhering to real-time constraints on a multitude of heterogenous hardware platforms, not necessarily under their full control.

- They integrate a variety of sensor and communications interfaces, short- and long-range, into their activities. Interaction with the associated devices ranges from continuous, frequent and even intervals to event-driven and interactive. In consequence, neither purely automotive development approaches, nor the common telematics frameworks congruently fit the mission.

- They are deployed in unsafe, distributed environments outside of the operator's sphere of direct influence.

That leads us to the aims of this work.

1.2 Aims

With subject and context set, this work answers the following purposes.

1. *Introduce ETC to software systems*
   As sketched in the previous section, from a software point of view, the domain of ETC is similar, but not equal to related domains. With its use case and deployment considered mission-critical, so is the design approach and the differentiation from the parent domains of automotive systems and traffic telematics a sensitive issue. Lacking other dedicated sources, the defining aspects and properties of an ETC system respectively its distributed software components need to be examined and described as a foundation for systematic further reasoning.

2. *Manifest reliability and economy*
   Unreflected, these requirements are arbitrary and obvious for almost any software system. Consequently, they need refinement and specialization on the ETC domain. Especially the constraints introduced by mitigating mission-criticality to the handling of monetary values inhibits the liberal application of costly solutions: reliability is balanced against economy, demanding a detailed analysis of the original facilities actually essential in this case.

3. *Provide a specialized architectural framework*
   A detailed, domain-specific set of requirements can determine the tailored construction of a compatible software architecture. As we regard the ETC domain as archetype, i.e. a representative case study for an

original type of mission-critical telematics system, it is necessary to take measures to ensure broader applicability of the emerging solution and its elements. The resulting architecture may then provide a foundation for related scientific research and industrial development.

1.3 Approach and Structure

The previously described aims are successional in their results. Consistently, they determine the structure of this work, as the artifacts produced in one chapter are built upon in the following. While we present the primary results in chapter 4, the artifacts of the other chapters not only form their basis, but also represent secondary results that contribute to the field of software engineering and architectures themselves. They may be used as devices for conducted reasoning about specific views on telematics systems, like aspects, risks, employed programming languages and RTEs, and platforms.

Chapter 2 – Establishing the Domain of ETC Software Systems first describes the details of the parent domains *automotive* and *traffic telematics* relevant to this work and gives an overview of their respective system setups and software development approaches to clarify the differentiation from ETC. The chapter then introduces toll collection fundamentals – defining generic concepts and processes, automation – and narrows the focus on the ETC type and component representatively treated here; a GNSS/CN thick client.

The essential and characteristic aspects of ETC systems, i.e. concerns cross-cutting the software architecture and application, are introduced along with a generic business process of an ETC OBE software that details activities to be implemented and processed for operations.

Original artifacts: classification of telematics deployments, a software-specific view on ETC, defining aspects, generic business logics of the distributed component.

Chapter 3 – Substantiating Domain-specific Reliability and Economy describes a comprehensive refinement of the initial requirements. Starting with a systematic approach to reliability, the chapter submits the previously introduced deployment to a fault tree analysis (FTA), identifying potential problems and isolating risks. From the results, we derive a set of architectural requirements directed toward a stable system. The elaborations on cost-efficiency on the other hand are constructive: the chapter presents three approaches to optimized utilization of the costly ETC infrastructure; that is, leading to what requirements should the software fulfill in detail to enable these scenarios.

After completing the requirements set – detailing how to achieve a domain-specific, balanced interpretation of reliability and economy – we have gained the mold of a high-integrity telematics software, with a strong emphasis on state machine-based structure and behavior. In addition, we examine its development process itself to pursue a holistic view and further support its qualities with constraints to the model and programming language, prominently enforcing static resolvability of structures and reasonably selecting a subset of Ada.

Original artifacts: a scheme for risk evaluation of mission-critical telematics software, a concept for smart card-based interoperability, a set of domain-specific requirements for a high-integrity architecture, complementing constraints for a corresponding model and source code.

Chapter 4 – Fundamental HIRTE Patterns describes the primary results of this work. In preparation for their specification, the chapter introduces behavioral and structural stereotypes to enhance the UML accordingly. The *distributable state machine fragment* offers a substantial contribution, as it is preceded by a formal extension of the state machine concept: the *self-contained state*.

With four patterns, conforming to the previously given requirements, we present the *HIRTE* – a *High Integrity Run-Time Environment* for mission-critical telematics applications, especially ETC. Instead of constructing a dedicated, monolithic architecture solution for our domain, the separation into patterns benefits reusability, conceptual evolution and generally attains a higher degree of universality.

Two of the patterns provide the blueprints for the design of high-integrity components for the realization of services in the context of an OBE application. The *Statically Resolvable State Machine* (SRSM) pattern specifies an executable hierarchical state automaton structure that completely avoids polymorphy, dispatching and similar dynamic techniques and mechanisms. The *Virtual Control Unit* (VCU) pattern offers a virtual machine for *Interpretable State Machine Code* (ISMC) that represents state machines as a sequence of opcodes.

The other two patterns complement a component-based architecture. The *State Tracer* pattern specifies a structure and mechanism to trace a system's state configuration quasi real-time. The *Distributable State Machine Fragment* (DSMF) pattern applies the previously defined stereotype to an executable state automaton, enabling it to delegate declared parts to remote interpreters for external processing.

All HIRTE patterns feature reference implementations in Ada, conforming to the high-integrity constraints of the given domain. These sources – instantiating a proof of concept to begin with – are fit to practically validate the concept, establish a framework or for further development and extension.

Original artifacts: the concept of self-contained states, SRSM, VCU, DSMF, State Tracer patterns and high-integrity reference implementations.

Chapter 5 – HIRTE Application Architecture Patterns sets the fundamental HIRTE patterns of the previous chapter in the context of an application architecture, thus giving them coherence and further purpose in roles. To this end, three successive patterns lower the abstract view to a concrete level.

As a high level view, the *framework* pattern illustrates the relationships of the HIRTE elements when arranged into a framework for OBE applications. The *implementation* pattern refines the element roles, details their associations and proposes alternative realizations of module integrity. The work is concluded with an *application* pattern that sketches actual ETC OBE services realized by HIRTE components of 4, coming full circle from the previous chapters with an architecture for a software that corresponds to the introduced aspects of 2 and fulfills the domain-specific variation of reliability and economy, the requirements of 3.

Additionally, the application section proposes an original concept of component separation between a regular OBE and secure module RTE. This is based on the interoperability approach of 3.1.3.3, facilitated by the HIRTE, especially DSMFs.

Original artifacts: the HIRTE concept, implementation and application patterns.

Chapter 6 – Conclusion and Outlook discusses approach and results of this work, presenting a structured evaluation. Besides opportunities for further research, open issues and a general outlook on the topics touched, we consider the potential practical impact of our results.

Legal Disclaimer This work benefits from now twelve years of continuous development and operation experience with a complex, large-scale GNSS/CN ETC system. However,

- this work merely employs ETC as an archetype of a scientifically rewarding, mission-critical telematics use case, suited for high-integrity software engineering,
- where this work elaborates on ETC, it establishes the general domain, describes generic deployments, components and interfaces, not any specific system currently in operation,
- discussed examples of (sub-)systems and (sub-)processes reference only publicly available sources,
- fault and risk analyses are based on the generic ETC architecture introduced by this work and do not reproduce any actual incidents, past or present,
- methods and approaches of this work do not allow any inference on any past or present methods and approaches used in the development of an ETC system currently in operation, neither how something was/is done, nor how somethin was/is not done.

Two

Establishing the Domain of ETC Software Systems

... in which we define the aspects of electronic toll collection, their deployment attributes and alternatives. A discussion of the ETC parent domains enables a characterization of the ETC software domain and its differentiation. For the further elaboration on the ETC software architecture, technical and operational preconditions are given. Generic business logics provide a starting point for the architecture requirements analysis.

To present a valid software architecture for ETC systems, this work first has to stake out their domain. On a general level, we find ETC a subdomain of both *embedded automotive software* and *traffic telematics*. Telematics in an automotive environment today seem to be common and established enough, the introduction of telematics services into a vehicle seemingly natural and effortless. After all, modern cars already represent "computers on wheels".

Interestingly, while a large number of applications are available – in different stages of product maturity –, as we will see, some characteristics of the domains seem to contradict each other. This is of little concern if we deal with non-critical services on dedicated and isolated platforms, e.g. a navigation system that updates its maps via GSM, but is installed into the car with its own sensors and dedicated CAN. There are scenarios however, where a tighter integration and/or higher criticality of the software brings these contradictions to the front. ETC is such a scenario, producing very specific requirements for an implementation, as will be shown later.

2.1 Related Domains

2.1.1 Automotive

2.1.1.1 Car Control Circuit

The automotive software technology domain traditionally focuses on a level close to the hardware. Engine, brake or chassis management (cmp. [Bo022], [SZ06], 2.1 ff) compose *transforming* functions with limited and clearly defined interactivity. They are integral parts of a control circuit system comprised of driver, vehicle and environment (cmp. fig. 2.1). The vehicle's setpoint devices translate driver's directions W^* to W for the network of controllers, which feed actuators the processed values U. Accordingly, the actuators implement the nominal directions Y on the road, with the result X measured by sensors in turn providing the controllers with actual feedback data R for comparison and adjustment. The environment introduces a disturbance variable Z.

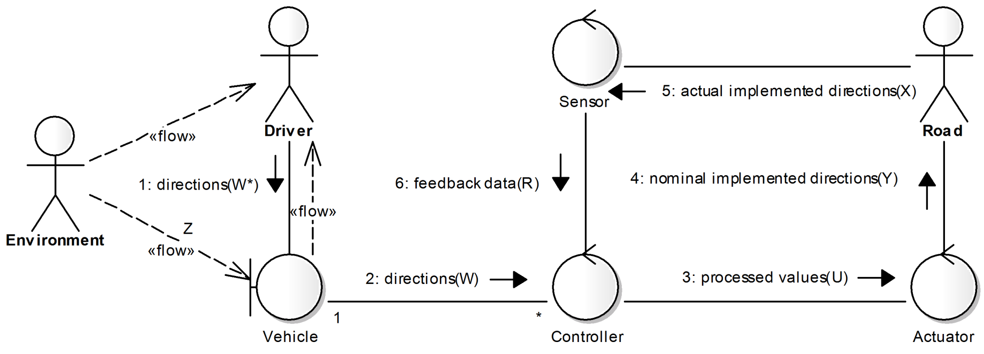


Figure 2.1: Generic Car Control Circuit

Note that, naturally, the illustrated circuit is abstract in that W is not necessarily a direct, driver-induced input for any controller. As we will see, controllers are organized in networks. Thus, interaction is also realized between controllers without some actual user interface, with the actor driver only indirectly influencing the process.

real-time constraints

As a basic principle, interactions are submitted to *hard real-time* constraints (cmp. e.g. [Li00]): the timespan between receiving a signal and sending the result as well as the timespan to communicate a signal from one controller to another are subjects to defined deadlines. Timescales in this context are measured in the magnitude of micro- and milliseconds. Many automotive subsystems like brakes, steering etc. are clearly classified as *mission-critical*, i.e. the safety of human lives or at least substantial material assets depends on their faultless performance. Consequently, violating the constraints is not acceptable regardless of the system load. An automotive development process requires explicit verification of the invariant validity of these constraints.

mission-critical systems

transforming, intra-car isolated components

For the following considerations, a *transforming component* accepts sets of numerical signals from various types and numbers of sensors. These signals are first obtained by sampling[1] a time- and value-continuous data stream (i.e. a signal S of W or R yields a unique value $S(t)$ for any time t) conforming to a defined sampling rate dT, in order to map it to a time-discrete sequence of values. dT may be constant or dynamic, e.g. in proportion to speed, and of differing value for different component inputs. Thus, the result finds time- and also value-discrete readouts, as the signals are digitally encoded by a limited number of bits, the range of the corresponding input variable. Likewise, the output value sequence U can be mapped from time- and value-discrete to continuous after processing by the component, if the receiver expects analog input.

transforming components cont.

For the software component processing, we assume a *stateless, functional relationship* between the input and output values. Incoming value sequences are transformed directly according to the implemented function algorithm, without input set i influencing the computation of set $i + 1$. Examples for the description of continuous relations can take the form of differential equations, or difference equations in the discrete case. Production of the output sequences has to conform to real-time constraints (s.a.); the computing capacity of the component has to match dT (dT_{min} in the dynamic case), so that no signal buffers are necessary. Beyond the intra-car interfaces, the components are, aside from maintenance purposes, isolated from the outside world during regular operations.

2.1.1.2 ECUs and Busses

The hardware platforms of automotive software applications commonly take the form of *Electronic Control Units* (ECUs, [SZ06], 2.3.1, [Bo01]), programmable microcontrollers connected by bus systems like CAN (*Controller Area Network*, [ISO94]), LIN (*Local Interconnect Network*, [LI06]), MOST (*Media Oriented Systems Transport*, [MO06]) or FlexRay ([Fl05]). For our purposes, we can categorize busses into *highspeed real-time* (CAN, Flexray), *lowspeed* (LIN) and *multimedia* (MOST). We will discuss bus characteristics in the light of ETC in chapter 3.1.3.2.

ECU characteristics

From our software point of view, an ECU as a "computer-on-a-chip" represents an adequate computing device on which to run programs on: outfitted with a processor (4, 8, 16 or 32 bit), RAM for data storage, ROM (e.g. EEPROM or Flash) for code/parameter storage and input/output interfaces, e.g. to/from sensors and busses. Additionally, embedded devices are often outfitted with a watchdog that checks the system's responsiveness at regular intervals and restarts it if necessary (and sensible from the application's perspective).

[1]The signal's origin is generally assumed to be analog. Whether the actual sampling is performed by the sensor module itself or the preprocessor of a microcontroller component is not relevant in this context.

Car infotainment platforms even approach the computing power of regular PCs. With that, they mark an evolution from the established transforming ("input-processing-output", s.a.), stateless unit to a reactive module with persistent memory and processing states (cmp. 2.1.2).

2.1.1.3 Vehicle Subsystems

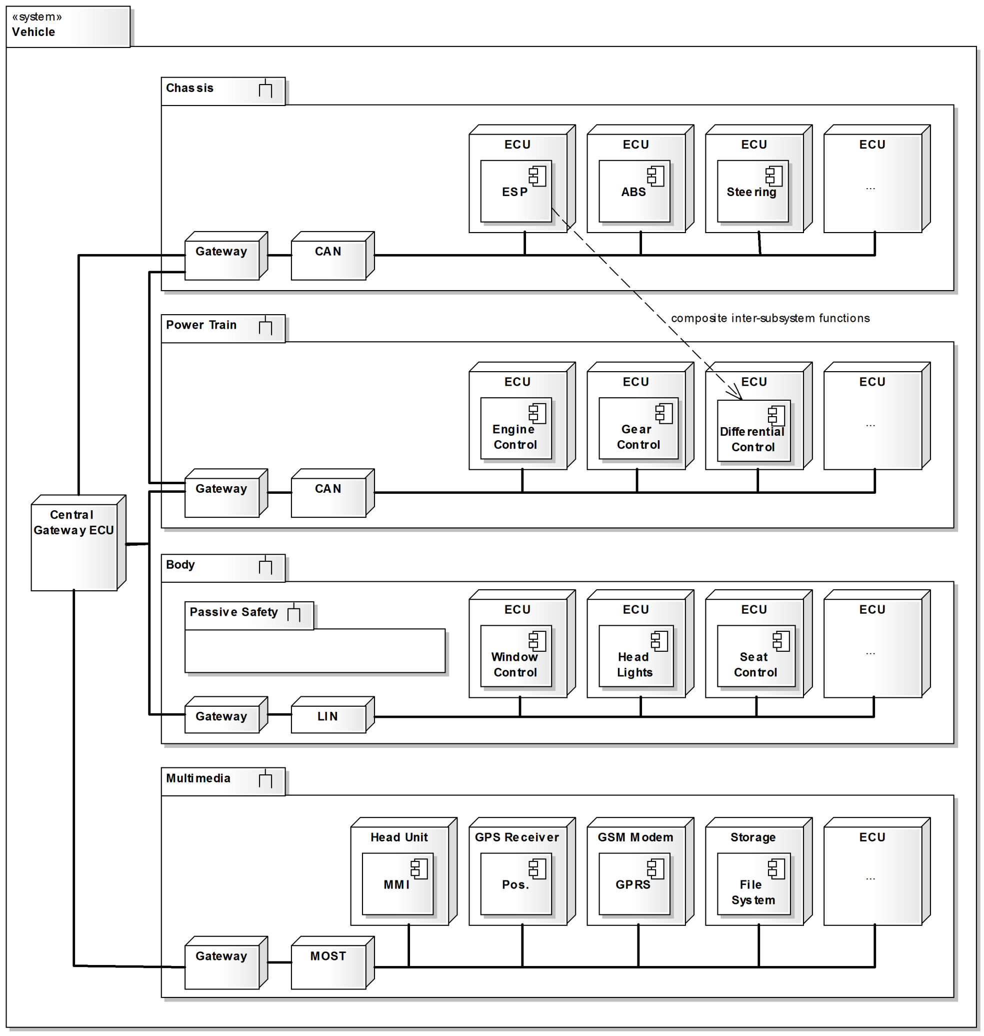


Figure 2.2: Automotive Subsystems illustrated

Connected by the different busses, the ECUs form electronic vehicle subsystems. Fig. 2.2 illustrates such a deployment. The resulting network provides a logical layer to implement complex, composite functions. Note the simplification in that certain applications, in addition to the also omitted sensors, imply integration of more than one ECU in a subsystem. Their specific capacities and characteristics depend on the tasks they have to realize in the overall system *car*:

Chassis – finds software functions of varying complexity. They range from plain translation, e.g. in the case of the handbrake or tire pressure monitoring, to brake-/steer-by-wire systems and integrated applications, e.g. safety measures like ABS (Anti-lock Braking System), ESC (Electronic Stability Control) and SBC (Sensotronic Brake Control, [Bo02]), aggregating the previous. The currently still prevalent bus is CAN, serial, with priority-based, event-driven bus allocation for short messages (up to 8 bytes of data), resulting in a maximum transmission rate of 1 Mbit/s.

Power Train – implements comparably few, but complex functions of engine and gear control. Correspondingly, here the number of deployed ECUs is rather low, with high-performance units for real-time processing of the setpoint input by accelerator and shifted gear, complemented by numerous sensors, e.g. air mass, temperature, throttle position. Here also, the CAN bus is most common. However, we find an aspiring successor in FlexRay, with time-based exclusive bus allocation for messages of up to 254 bytes of data and a maximum transmission rate of 10 Mbit/s.

Body – is subdivided into the critical passive safety functions (airbag control with seat occupancy detection, roll-over bars), and the less critical convenience functions, like window controls, rain sensors, air conditioning etc. The current generation of cars features the highest number of ECUs and software functions in the body, but with a low degree of inter-ECU integration compared to the above subsystems, and less synchronisation parameters between its units. Computing capacities are also minor, reflected in the slow LIN bus: up to 20 kbit/s, between 1 and 8 bytes of data per message (fixed for any message identifier on setup), deterministic, signal-based allocation.

Multimedia – represents a departure from the embedded approach of realizing existing car-specific mechanisms in ECUs and software, or creating additional functions by composing them. Instead the subsystem introduces functions from other domains into the vehicle: video/audio, telecommunications and navigation. In consequence, software-technologically it anticipates section 2.1.2, as most of the in-car telematics components currently can be found in the context of the multimedia subsystem. Processing capacity and software complexity are naturally high, with the MOST bus providing synchronous streaming as well as asynchronous packet transmission capabilities with an overall maximum bandwidth of 50 Mbit/s.

ECU integration and concurrency

With e.g. ESC or the Adaptive Cruise Control (ACC), we find composite software functions that cannot be assigned strictly to one subsystem. This inter-subsystem integration of components is enabled by *gateways*, with a central gateway commonly used for diagnostics and maintenance. Even with the general low-level, hardware-close programming associated with microcontroller software, the consequent sharing of resources becomes critical when it goes beyond mere readouts of sensors. Concurrent access to actuators – e.g. ACC overriding driver induced actions on converging on another vehicle – demands adequate, strict formalisms of software specification, implementation and verification concerning scheduling, priority management, mutual exclusion etc.

2.1.1.4 Development Approach

hardware and programming

The car-fitted ECUs have to be very intimately known hardware components. Imperative precondition for changing the manufacturer or series is a thorough, most accurate certification process. This is complemented by the fact that a vehicle's product life cycle covers around three years of development, seven years of production and ten to 15 years of operation and service (cmp. e.g. [SZ06], 1.4.2.3). Thus, implementing software hard real-time behavior is significantly facilitated by the fact that the program can rely on the predefined, exact behavior of its run-time environment. Integration of software and hardware is therefore tight, with little or no abstraction layer between program and machine.

techniques and methods

The implementations as elements of the car need to exhibit an obligatory, validable stability and compliance to a range of regulations, e.g. concerning safety, ecology; pollutant emissions. Fundamental standards applied to automotive systems are represented by DIN 19250 [DIN89] and IEC 61508 [IEC98]. Consequently, the software is submitted to strict specifications and formal constraints. Development processes – themselves subject to certification – are therefore dominated by e.g. the Mathworks environment ([Ma07], with Matlab/Simulink and Stateflow) and MISRA C [MI04] as preferred language subset. Conformance to run-time and memory restrictions is implied by usually very limited computing resources, resulting in static, safe programming techniques, of which we will learn more in chapter 3.2.

the question for automotive software frameworks

Production processes of component suppliers in the automotive industrie are highly standardized and formalized in every way from specification to acceptance and commissioning to the manufacturer. Given the criticality of the technology both for human lifes (ecology and operational risks, safety issues) and the economy (notably in Germany), this is comprehensible. Numerous past and recent publications stressed the fact that software as a car component still gains importance. Coherently we have to ask the question for a fundamental, widely adapted automotive software standard beyond the established practices mentioned above.

basic aims of frameworks

Over the years, a number of significant endeavours to establish a standard framework for automotive software modules were conducted, in each case with the participation of all major automotive players. The basic aims were similar in all cases:

- provide an accepted run-time environment by introducing a middleware between ECU software and hardware or OS (where present), or substituting[2] the OS,
- abstract and unify the communications between ECUs with a common mechanism (e.g. layer, bus) and
- generally define an approved interface for ECU applications with the corresponding set of implementation rules,
- in order to facilitate component portability over vehicle lines and revisions, especially important due to the long product lifecycles (see above) and the nevertheless evolutional market of ECU hardware.

The results – e.g. *Offene Systeme für die Elektronik im Kraftfahrzeug / Vehicle Distributed eXecutive* (OSEK/VDX, [OS05]), *Herstellerinitiative Software* (HIS, [HI04]) and the most recent candidate *Automotive Open Systems Architecture* (AUTOSAR, [AU06]) – have to prove themselves longterm beyond pilot projects and non-mission-critical applications. We discuss some of their specifics related to this work in chapter 3.1.3.2. Here, we assume a more general perspective.

potential practical framework contradictions

From the software view, all approaches share common potential flaws: given the criticality concerning both risks and proprietary, competition-relevant knowledge, it could be detrimental if the standard's RTE implementations would be supplied by any single source. Instead, to ensure the necessary complete control over the mechanisms of the supplied item, each automotive component supplier might effectively be coerced into realizing its own interpretation. The stated increased importance of software in cars implies competition-relevant assets in the programs, which have to be protected. In turn, this encourages the tendency for proprietary extensions, optimizations to gain technological advantages. Furthermore, in a way, the resulting middleware is contradictory at least to some established paradigms – efficient, lean programming close to the hardware – of the automotive software domain. Whether or not an approach like AUTOSAR eventually leads to cost-savings by standardization or more expensive electronics due to high complexity and performance requirements needs to be proven. In any case, the framework approaches do not mitigate, but merely strive to standardize the described domain-specific characteristics of automotive software. With the possible exception of tight hardware-orientation as stated above, the high-integrity implementation constraints stay valid.

[2]Actually, due to the often very limited resources of the automotive environment, in the given context OS may equal framework.

2.1.2 Traffic Telematics

telematics narrowed down to traffic applications

Traffic telematics as a subdomain of the general telematics encompass a variety of use cases, e.g. related to public transportation, individual or commercial road, air, rail and sea traffic. Government-initiated traffic telematics endeavours are often related to ecology or safety. As a general rule, they aim at influencing commercial and/or individual mobility habits, either by punishing malpractice (e.g. by tolling and fees, restrictions) or rewarding the desired behavior (e.g. optimized access to infrastructure, tax savings). Safety applications find e.g. automatic emergency calls (*eCall*, [eC06]) or road hazard alerts deduced from floating car data (*FCD*). Commercial, market-driven products can be subdivided into business applications, e.g. for the logistics domain, and convenience or entertainment.

2.1.2.1 A Telematics Infrastructure

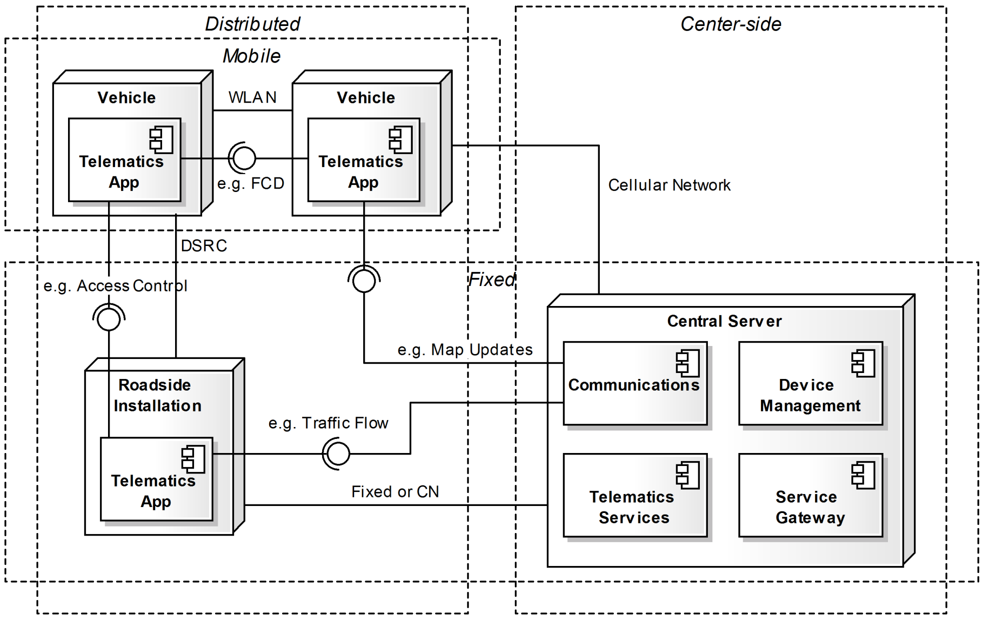


Figure 2.3: Traffic Telematics Infrastructure Overview

Fig. 2.3 illustrates a generic infrastructure deployment from the perspective of this work. Communicating software components act in standard and embedded computer environments, joined by a variety of interfaces. Constituting their operational effort and costs (also cmp. [Ste03]), we classify components conforming to their hardware's *locality* – center-side or distributed – and *mode of deployment* – fixed or mobile –, as these attributes and their determining factors strongly influence behavior, upkeep, maintenance and interconnection requirements.

locality and mode of deployment

Center-side systems represent components running on server infrastructure. Determining quantifiable quality of service (QoS) factors are given by availability, the type and number of parallel connections to distributed components and the performance concerning transactions per time unit; average or during peaks.

Functionally, communications implement the basic protocols of distributed transactions over fixed lines or a cellular network. If more than one provider is utilizing the servers, a gateway dispatches messages to and from the respective service-related instances. Device management handles the general, not necessarily service-specific remote provisioning and maintenance of distributed components.

Distributed systems can be deployed in two ways:

- **fixed** – encompass roadside installations like traffic flow measurement units, signal beacons and access control mechanisms, e.g. for harbors or large-scale construction sites. These embedded devices are outfitted with an usually very limited array of sensors and interfaces corresponding to the implemented service they provide. For device to vehicle interaction, we find *Dedicated Short Range Communications* (DSRC) via microwave or infrared. Long range interfaces depend on the site accessability of the installation, with some fixed line in the case of close proximity to existing network infrastructure and CN for more remote sites. The same differentiation is relevant for power supply and the corresponding costs. Additional determining factors of fixed systems are safety (the installation must not pose a hazard for passing traffic) and security measures of the hardware and software, e.g. protection against tampering.
- **mobile** – in our context designate automotive *On Board Equipment* (OBE) composed of a run-time environment for software with a processor, memory, interfaces to the user, sensors and communications. Evident defining factors of these components are the processing capabilities and memory size and types in relation to the software's complexity regarding data structures and concurrency, as well as the associated features of the user interface. These range from plain on/off-switches, e.g. in the case of an area access tag, to the color touch screens of navigation systems.

  A very important effort and cost driver is the mode of in-car installation: integrated/ECU as part of a subsystem (cmp. 2.1.1.3), connected to a subsystem (e.g. manufacturer pre-installed telematics), fixed installation with dedicated bus (aftermarket, e.g. German Toll Collect OBU) or cradled with no connection to vehicle electronics at all (e.g. current off-the-shelf navigation solutions).

This also has a high impact on software component interfacing, and conversely, as we will discuss in 3.1.3.2. Depending on the approach of integration of the telematics application in the vehicle, the question for communication protocol standards and accessability of the vehicle ECUs arises. Consequently, it further characterizes the respective component in its transactions with its environment.

thin and thick clients

Additionally (and not shown in the illustration), we have to distinguish between *thin* and *thick*[3] mobile clients. This refers to the complexity of software processes run on the mobile equipment, i.e. the division of computation workload the respective service requires between center (off-board processing) and distributed (on-board processing) components. It directly affects hardware costs, generally more critical on the distributed side due to specialized elements and higher maintenance effort. In many cases, it also has an impact on communications, as it may be feasible to compensate for mobile computing capacity by transferring data for central processing and replying the result. Here, a cost-benefit analysis of mobile equipment versus e.g. cellular network services and quality (also cmp. next section) is required. An example can be found in off-board routing approaches, compared to the common on-board solutions, to enable navigation on low-end mobile phones.

point of provision

Besides the above focus on processing, a final, alternative view on the characterization of the telematics service is the *point of provision*, i.e. whether the service's output is consumed on the road or centrally.

Central provision – distributed components collect data to feed a center service; e.g. traffic control systems.

Distributed provision – a central server feeds distributed components; e.g. GSM-updated navigation solutions.

Strictly distributed provision – the service is instantiated by ad-hoc networks of mobile components; e.g. FCD hazard warnings.

Combined provision – the data is circulated and refined between mobile components and central servers. Both sides introduce data into the service; e.g. disposition services for approaching trucks in large cargo transport centers, providing both optimized scheduling to clients as well as optimized capacity utilization for the center.

In the case of a commercial service, this influences the technical architecture as well as the business model in regard to payment and billing approaches.

[3][MVW08] further distincts *slim* and *smart* clients. Beyond a very specific context, we regard this as arbitrary, as it should be obvious that there can be numerous variants between the extremes of thin and thick client realization.

2.1.2.2 Communications and Interfaces

As mentioned above, a telematics infrastructure may feature interfaces for short and long range interactions, depending on the implemented use cases. In this section, we will not regard intra-vehicle interactions, as the automotive busses are discussed in the context of 2.1.1 and 3.1.3.2.

Table 2.1 depicts established and potential solutions for component communications, beyond the vehicle network, from a software applications's point of view[4]. Note that every case implies data and time for protocol overhead processing, diminishing the capacity for net user data.

Critical attributes of communications should cover the common parameters of quality of service, but for a telematics application additional factors have to be taken into consideration.

availability … temporal

All of the service's communications interfaces have to be analyzed for *temporal and local availability.* Referring to time, this information has to be derived from the interface technology's specification, field tests or it may be guaranteed by the vendor. It can be expressed by a percentage associated with a time interval or a discrete event (e.g. data update necessary) and should approximate the maximum likelihood for failure during the operating time or for failure of the event, respectively.

… local

Referring to locations, either the actual geographical regions have to be specified, or types of areas should be classified corresponding to the local communication service coverage (e.g. high availability in metropolitan areas). Furthermore, an area has to be analyzed for interfering structures or signal sources, which could degrade availability. Besides long range communications, also the internal interfaces of the mobile application can be of interest: in a multi-process environment, the availability of a commonly used communications interface may be determined by assigning priorities to the collaborating processes.

While today's GSM networks usually have a quite complete coverage, it is still possible that the vehicle discovers an area without cells. As it is, an important question for applications on mobile devices concerns the defined reactions to the temporary failure of communications interfaces – this situation may arise frequently and should not be critical for the overall stability and functioning of the service implemented by a component.

throughput and transfer rates

As communication occurs, in most cases there is limited time to complete the transmission. To ensure efficient transactions, the underlying technology has to provide sufficient *throughput* or *transfer rates.* We interpret throughput in a rather loose sense, not only describing transfer rates of continuous connections like bits per second, but also the length of data encodable in a discrete

[4]I.e. we disregard physical or signal-engineering issues and do not differentiate between technologies, protocols and their ISO OSI layer mapping. A corresponding wrapper is presumed.

Technology	**Throughput and Comment**	**Range or Coverage**
Wireless LAN (IEEE802.11)	Approx. 11 Mbit/s. Vehicle to vehicle and vehicle to roadside. A number of proprietary adaptions for the automotive environment exist (e.g. [C2C08]).	Currently approx. 150 m.
WAVE (Wireless Access in the Vehicular Environment, IEEE802.11p)	Adaption of WLAN for data exchange between passing vehicles up to 500 km/h relative speed, up to 27 Mbit/s gross. Currently a draft.	Supposedly up to 1,000 m.
Dedicated Short Range Communications, Infrared (e.g. ISO 21214:2006)	Currently maximum of 500 bytes at a passing speed of 150 km/h, vehicle to roadside.	Approx. 25 m.
GSM Short Message Service (GSM 03.40)	140 bytes of user data per message. Message based protocol of GSM.	Coverage can be considered sufficient for any telematics service in western Europe.
GSM Bearer Service 26 (GSM 02.02)	Asynchronous, connection-oriented transmission of up to 9,600 bit/s. Establishing the connection may take some seconds.	Dito.
GSM General Packet Radio Service (GSM 07.60, 09.61)	Packet-oriented transmission of approx. 40 kbit/s downlink and 13 kbit/s uplink with a typical access time of .5 to 1.0 sec.	While based on GSM infrastructure, still requires cell modifications in some areas.
Universal Mobile Telecommunications System (3GPP TS 22.100)	Allows packet-oriented uplink and downlink between approx. 384 kbit/s and 1.8 Mbit/s.	Currently insufficient coverage except for urban telematics services.

Table 2.1: Example Media of Communications in a Telematics System

message like in SMS, which can be sent completely (i.e. it is not necessary to split it up) in a specified frequency (e.g. one message per second).

throughput and availability

Especially for the evaluation of a maintained connection (like e.g. BS 26) between mobile component and central system it is important to correlate the throughput with availability: transfer rates may drop dramatically in the case of degrading availability, making it necessary to identify the corresponding situations and their relevance for the service in question. A telematics application should keep the data to exchange with a moving unit as short as possible. If the mobile component requires larger updates of its operating data, eventually SMS may become insufficient. As BS 26 connections tend to break down while moving (resulting in less available throughput and the necessity of a protocol to enable partial transfer and continuation), either a packet-oriented service (like GPRS) has to be used in these cases, or the data transfer should only be initiated when the component is stationary.

interoperability of interfaces

The acceptance of a telematics service can be influenced strongly by the question of *interoperability*, concerning both application-internal and external communications. If the mobile application communicates with its underlying OS and hardware over a specified interface (e.g. an API) or middleware that may be implemented on a number of different platforms, the service can be deployed on a wider range of devices. External interoperability has to deal with communicating with infrastructure outside of the service provider's proprietary systems. An implementation might have to take a future Galileo into consideration for positioning, in addition to or as substitution for the current GPS. GSM services may be used for over the air data exchange because of the currently quite complete coverage of Europe, although the application should interoperate with different GSM networks to ensure regional fallback where necessary – in the case that one network has better coverage in certain areas than another – and enable roaming in other countries. As newly available and future cellular phone technologies (next generation UMTS etc.) will give access to higher data transfer rates, the component's interfaces have to be prepared to interoperate with these correspondingly. Furthermore, a number of telematics services rely on roadside infrastructure, like e.g. DSRC beacons. Accordingly, a system integrating beacons should be able to communicate short range via both microwave and infrared.

2.1.2.3 Development Approach

informing systems

The current market of vehicular telematics applications reveals a prevalence of *informing systems*: multimedia, navigation, floating car data – generally, with few exceptions like the eCall, designated non-mission-critical. An informing system's output may range from merely convenient to cost-effective, e.g. in the case of a logistics disposition solution. However, no associated actor is dependent exclusively on its performance. Real-time constraints are soft, corrections and retries of many operations are possible. If a navigation's routing fails, the driver has to find an alternative way; an approaching delivery to a freight harbor that failed to register automatically has to and will wait for a free unloading slot.

interactive components

In contrast to the deterministric transformation of 2.1.1.1, interfaces to these implementations are open, variable and very interactive, with the user as well as outside service servers being able to access and interact with the software at any time. [Ku05] gives a similar concept of *reactive* systems continually interacting with their environments. In a strict sense, this term seems appropriate, as even activities initiated by the respective component may be the reaction to some internal event, e.g a time trigger. For our context, we need to emphasize the interactions with other components like depicted in the above telematics infrastructure, and the component's ability to initiate activities as a proactive process. Thus, we will designate this kind of component *interactive.*

standard requirements, relaxed constraints

Due to the informing character of the services, requirements concerning reliability are rather loose, or at least not significantly higher than for any other commercial application. With the introduction of lavish graphical and voice-controlled user interfaces into the car console, the traditional resource restrictions of embedded systems in these cases became a thing of the past. A direct result was the departure from restricting high-integrity development environments and languages (cmp. 2.1.1.4) to more common approaches like Java (also cmp. 3.2.2). In the process, practical experience finds many cases where features like dynamic memory management, dynamic binding, polymorphy, garbage collection and complex libraries were introduced unadjusted into an embedded environment.

dynamically evolving platforms

Distributed hardware platforms for telematics services are highly dynamic in many regards. Not only do device series and models change quickly to attune to market demands, the developer also has to consider different kinds of devices to be fitted into the vehicle, from proprietary OBE to mobile phones. In order to reduce development costs and serve the consumer in an economical fashion, this called for abstraction layers to ensure flexibility and keep software porting efforts over series in check.

run-time environment examples

Telematics software products thus integrate with a specific OS and, if applicable, framework rather than hardware platform, e.g. processor family. Device abstraction and application environments are provided by correspondingly complex products like *Windows Automotive* [Mi07] with Fiat's *Blue&Me.* The complexity and architecture of these solutions is much closer to the common desktop/server OS than the established embedded minimal RTEs with lightweight memory footprints and microkernels, e.g. *QNX* (cmp. [Hi92]). Windows Automotive even provides a Direct 3D graphics API. Most of these approaches have the dynamic mechanisms mentioned above in common, inducing non-deterministic behavior and generally impeding hard real-time conformance.

2.2 Electronic Toll Collection (ETC)

2.2.1 Toll Collection Fundamentals

Toll Collection generally aims at the pricing and charging of traffic infrastructure access: roads, areas, tunnels, bridges. In contrast to non-discriminating approaches like taxes on vehicles and gasoline, its levy is based on the actual usage of specific elements. The user, commonly the driver of some vehicle, discharges a defined amount of money for accessing the corresponding infrastructure.

Note that definitions and descriptions of this work may differ from [ISO03]. As we focus on the software architecture of ETC OBE, our priority is clarity in the given context rather than adherence to a standard with a significantly wider perspective.

closed and open systems

The traffic infrastructure managed by a toll collection system is either represented by a single element, e.g. in the case of a major tunnel, or some aggregation, e.g. a directed graph in the case of a motorway network. *Closed* systems (cmp. e.g. [PB06]) detect entry and exit of the vehicle to and from the element or aggregated structure. The amount due is then derived as flat/fixed rate, or from the time or distance travelled inside the system. *Open* solutions register each discrete element access by the vehicle, allowing for prompt charging.

toll atoms

In the following, we will designate a single, not further resoluble infrastructure element processed by a tolling system a *toll atom*. They are described by

$$TA \subseteq L_{TA} \times M_{TA}.$$

Fundamentally, toll atoms are attributed with a location L_{TA} and measurement M_{TA} component. The former specifies a fixed location like a road segment enclosed by some entry and exit point, the latter any minimum, i.e. atomic duration or distance measurement unit incremented when using the associated location. The actual localized reference is system-specific: it may be given by an enclosing traverse (area), linked vectors (road segment), both for GNSS detection, an ID of a segment's roadside installation for DSRC detection, or any proprietary data structure (e.g. a virtual tolling spot defined by ID, coordinates and radius).

Example 1 (British London Congestion Charge) processes vehicle access for one day to inner city London:

$$TA_{LCC\ GB} = \{(<\mathit{downtown\ area}>, <1\ \mathit{day}>)\}.$$

Example 2 (Swiss *Leistungsabhängige Schwerverkehrsabgabe*) – records kilometer usage of trucks inside the borders of Switzerland:

$$TA_{LSVA\ CH} = \{(<\mathit{national\ territory}>, <1\ \mathit{km}>)\}.$$

Example 3 (French *Télépéage Inter-Sociétés Poids Lourds*) – checks vehicle entry and exit to/from selected subsets of road sections, managed by a multitude of TC operators:

$$(< motorway\ section >, < length\ in\ km >) \in TA_F.$$

Example 4 (German *Lkw-Maut*) – detects a vehicle's usage of discrete road segments (entry to exit ramps) with defined lengths:

$$(< motorway\ segment >, < length\ in\ km >) \in TA_D.$$

Note that this, similar to France, is a special case of distance based tolling in that it processes segments and adds up their lengths to obtain a reproducible measurement instead of directly measuring each kilometer.

tariff scheme

The pricing associated with a toll atom is described by a *tariff scheme.* The tariff scheme defines a mapping

$$Tariff \subseteq TA \times P_{vehicle} \times P_{driver} \times P_{time} \times P_{operator} \times T$$

that assigns a set of parameters to an amount of money T to levy, with

- TA the accessed toll atom,
- $P_{vehicle}$ the vehicle-specific parameters like type (truck, car, bike, other), number of axles, pollution/emission, weight class,
- P_{driver} the driver-specific parameters, e.g. concerning profession (police, paramedics) or status (e.g. disabled),
- P_{time} time-specific parameters describing periods, by day (e.g. holidays) or by hour segments (e.g. rush hour) during which T applies,
- $P_{operator}$ other, operator-specific parameters concerning roaming/interoperability agreements, discounts for modes of payment, voucher handling, credit accounts, combination with services or specific products.

The tariff scheme mapping may be established either explicitly, i.e. in the form of a matrix, or functionally, i.e. in algorithmic form.

Example 1 cont. for a regular car:

$$\begin{array}{lll} (ta_{LCC\ GB}, & < car >, & < regular >, \\ < 7\ am - 6\ pm >, & < 8.0\ £ > &) \\ \in Tariff_{LCC\ GB}. & & \end{array}$$

Example 2 cont. for vehicles over 3.5 tons, weight limit 40 tons:

$(ta_{LSVA\ CH},\quad <\textit{permissible max weight}>,$
$<\textit{average emission class}>,\quad <1.7\ \textit{ct per ton and kilometer}>)$
$\in \textit{Tariff}_{LSVA\ CH}.$

Example 3 cont. for large trucks, collected by a specific operator:

$(<\textit{Paris – Le Mans}>,\quad <\textit{class 4}>,$
$<\textit{Cofiroute}>\quad <51.10\ \textit{EUR}>)$
$\in \textit{Tariff}_F.$

Example 4 cont. for a comparably environmentally friendly truck with four axles:

$(<\textit{AS PB-Elsen – AS PB-Zentrum}>,\quad <4\ \textit{axles}>,$
$<\textit{locational class}>,\quad <\textit{temporal class}>,$
$<\textit{emission category A}>,\quad <0.34\ \textit{EUR}>)$
$\in \textit{Tariff}_D.$

Note that both locational and temporal classes are currently implemented but not activated.

tour

In the following, a *tour* designates a vehicle entering, travelling on and leaving the system's traffic infrastructure. Between entering and exiting, the vehicle uses any coherent sequence of toll atoms. The tour concept applies both for open and closed systems, with implicit or explicit entry/exit, respectively.

rating and pricing

A vehicle successively accessing the system's infrastructure on a tour accumulates references to toll atom elements and attributes them conforming to the tariff scheme[5], in effect *rating* and *pricing* the usage based on currently valid parameters. While some parameters are fixed for each tour, e.g. the vehicle type, others will or may change over time, like the number of axles with or without a trailer. However, generally, the pricing determines the costs for using the infrastructure and can technically be handled prior to, on, or subsequent to the actual access, depending on the system's flexibility and payment paradigm (see below).

charging and billing

The resulting set of tariff elements are thus loosely comparable to the telecommunications domain's *Call Detail Records* (CDRs). *Charging* complements them by an individual vehicle's unique identification, i.e. license plate code, associating the specifically priced elements with a user. Subsequently, the resulting sum of T (total or as subtotals) is assigned to a debitor and presented to the user. Whether this is considered still a part of charging or already *billing*, again depends on the system's paradigms and the actual business processes. In our view, billing not necessarily interacts with the driver but any debitor entity (e.g. a logistics contractor) decoupled from the user, whereas the result of charging may be imminently relevant for the current

[5]Note that this merely refers to an abstract principle of tolling, not the technical process.

	Flatrate	**Measured**
Prepaid	Fixed payment on booking or starting the tour.	Specification and payment of a tour on booking, or payment into an account to balance future measured charging.
Postpaid	Fixed payment on ending the tour.	Payment on ending the measured tour, or after receiving a bill of the measured tour.

Table 2.2: Tolling System Payment Paradigms

user/driver. If the mode of presentation of a charging amount is regulated by legal issues, it may have a strong impact on the technical realization options. In the case that charging has to enable the user to instantaneously validate or at least acknowledge the respective figure, i.e. it has to be reproducible at all times, it narrows the approaches down to – like in Germany – either prebooking (terminals, internet) of tours or real-time charging with rating and pricing handled on-board by the OBE.

payment

Charged sums are balanced by a *payment* transaction, i.e. the flow of money, of the amount due from debitor to operator. The actual sequence and realization of pricing, charging and payment steps depends on the payment paradigm established by the tolling system, as described in table 2.2. "Flatrate" corresponds to one atom or a fixed rate for any sequence of toll atoms of a tour. "Measured" tolls each atom of a tour consisting of more than one. Other tariff parameters may still apply in both cases. Independent from the implemented paradigm, the finalization of a non-flatrate tolling business process has to mediate an equilibrium of the

- *kilometers driven by the vehicle*,
- *kilometers measured by the OBE or declared by the user*,
- *kilometers charged and reported to the center*,
- *kilometers billed to clients* and
- *kilometers toll collected from clients* (cmp. [SKG08]).

A generic business process from identification to payment is described by fig. 2.4 for reference.

enforcement

Directly related to the pricing/charging approach and especially relevant in systems with mandatory participation, like governmentally introduced road tolling, is the *enforcement*. It encompasses measures to

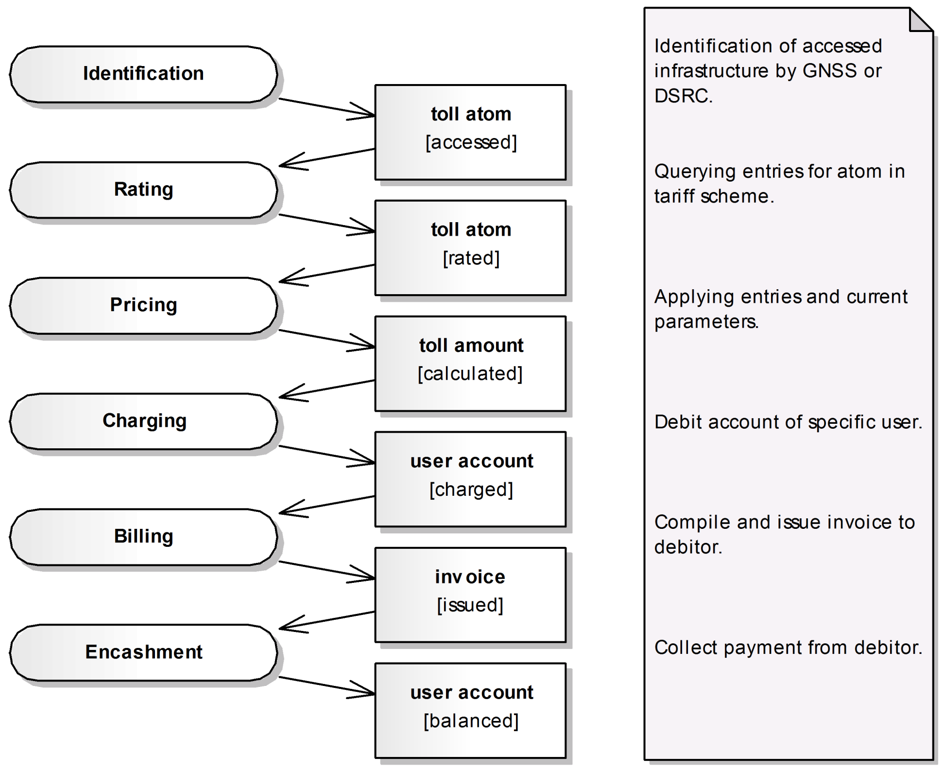

Identification
Rating
Pricing
Charging
Billing
Encashment
toll atom [accessed]
toll atom [rated]
toll amount [calculated]
user account [charged]
invoice [issued]
user account [balanced]
Identification of accessed infrastructure by GNSS or DSRC.
Querying entries for atom in tariff scheme.
Applying entries and current parameters.
Debit account of specific user.
Compile and issue invoice to debitor.
Collect payment from debitor.

Figure 2.4: From Toll Atom Identification to Payment

- check if a vehicle is obliged to participate in the tolling system,
- if so, check if a vehicle participates,
- if so, check if the parameters of participation in the tolling scheme correspond to the vehicle's configuration,
- if so, check if the pricing was correct and the charging accepted,

to ensure participation and enable fraud detection. On check failure, the enforcement may have to determine the reason, especially whether it is the effect of a technical failure or deliberate evasion. In the case of an intentional act, the vehicle may be physically barred from the traffic infrastructure, or the user may be fined. The efforts required of the enforcement implementation depend on the details and levels of the above checks necessary to validate conformance to the respective tariff scheme and charging modes. Enforcement mechanisms may from a technical view thus be plain, like the French tollgates, or sophisticated solutions, like the German enforcement bridges, outfitted with infrared, laser measurement units and video surveillance.

2.2.2 Automation of Toll Collection

general advantages

Electronic Toll Collection represents an evolution from the manual levy. As such, two advancements become immediately apparent. First, the automation of hitherto manual processes and the commonly associated cost-efficiency; second, the avoidance of interfering with the flow of traffic. Manned tollbooths are replaced by free-flow signal beacons or completely virtual entities, with the passing of a vehicle detected by OBE with GNSS, and no roadside infrastructure. Business logics are primarily implemented as software. In consequence, ETC is able to *refine* and *expand* the corresponding fundamental toll collection use cases (cmp. fig. 2.5).

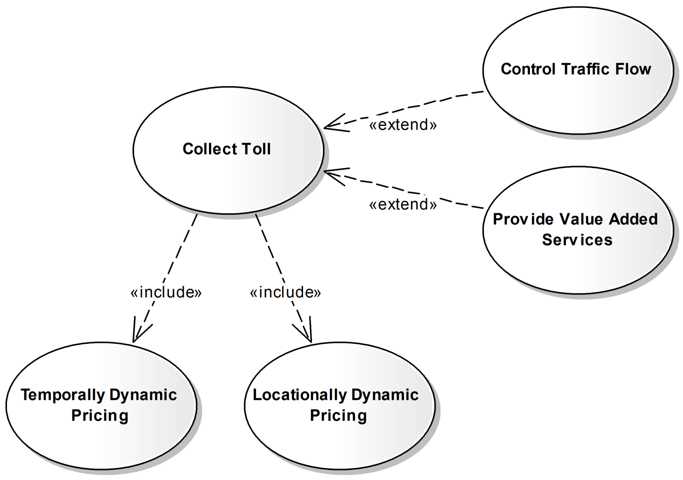

Control Traffic Flow
«extend»
Collect Toll
«extend»
Provide Value Added Services
«include»
«include»
Temporally Dynamic Pricing
Locationally Dynamic Pricing

Figure 2.5: Expanding Toll Collection Use Cases

TC refinements

Pricing may now be based on both more finely grained temporal and locational parameters that are kept dynamic, i.e. they change over time. These changes can be determined by fixed, periodical schedules or as reaction to irregular incidents, like construction sites or traffic jams, referring to specific road segments or hours. A manual solution would be challenged to implement these refinements with regard to pricing consistency and reproducibility.

Example scenario: a pricing adjustment effective from time t sees one vehicle passing a manual tollgate at $t - 1$ *min*, another at $t + 1$ *min*, after the price for the successive road segments increases due to the rush hour. The following traffic jam finds both vehicles on the same road segment at the same time. A manual approach would charge different prices after the preceding pricing. A GNSS solution could price the segment real-time and charge both users equally.

TC extensions

Beyond the vertical refinement, automation can help to extend a toll collection system horizontally, i.e. by additional use cases. To effectively control traffic flow, e.g. to achieve relief from environmental pollution, authorities require the means to convey such a directive. A tolling system in combination with an adjustable, dynamic tariff scheme may provide just that. As the increasing complexity of a tariff scheme implies growing operational efforts to determine parameter weights and costs and calculate prices, again, from a certain scale on – like truck tariffing in Germany – a manual solution would be unfit for implementation.

Another example for sensible ETC extension is the value added service (VAS), building on ETC-specific infrastructure for processing, storage and communications. VAS add user-supporting functionality to the ETC system that aim at optimizing its capacity utilization. We will discuss an ETC VAS approach in the context of economy in chapter 3.1.3.1.

2.2.3 Preconditions and Assumptions

Corresponding to the differentiation of the traffic telematics infrastructure in 2.1.2 we find three principal[6] ETC approaches to implement a tariff scheme as introduced above.

DSRC beacon-based – focuses on roadside infrastructure. Signals are exchanged between gantry-installed beacons and lightweight windshield OBUs that can be reduced to mere virtual tags. As there are no additionaly sensors, beacons are placed on any toll atom, e.g. like every motorway section in Austria. On passing, the OBE transmits the vehicle's ID, indicating the usage of the corresponding atom. Pricing and charging may be handled real-time on- or off-board, depending on the OBE capabilities.

Thin client OBE – focuses on communications and center-side systems. Business logics realized on distributed components and thus OBE processing capabilities are kept to a minimum. OBE collects personalized vehicle positions that are periodically transmitted to and evaluated off-board by central servers. The identified routes are priced according to the accessed toll atoms. Due to the characteristics of mobile communications concerning availability and throughput (also cmp. 2.1.2.2), there is a non-deterministic delay between usage, pricing and charging. Neither can the OBE guarantee the timely transmission of routes, nor can the center control the reachability of distributed components for replying with the receipts.

[6]We do not regard dedicated approaches for special, single toll atoms here, but universally deployable solutions.

Thick client OBE – focuses on complex software processes of distributed components. Business logics from infrastructure identification, i.e. locating, to pricing and charging are handled on-board. Real-time processing enables immediate user information, dynamic tariff adjustment and flexible enforcement, e.g by sensor bridges or mobile units. Charging data is transmitted periodically to and receipted by central systems. OBE requires sufficient locating sensors, interfaces, memory and processing resources to ensure a defined quality of service of its part of the ETC business processes. Expensive roadside infrastructure can be kept to a minimum.

focus on the thick client

As a subject of this work is an embedded architecture in an automotive environment, we will concentrate on the *thick client approach, more specifically GNSS/CN OBE.* It features the most complex processes of distributed software components. In consequence, results for a reliable and economical system implementation may at least partially be applied to less comprehensive variants. Note that this does not imply any bias toward the general usefulness or validity of any of the described solutions, but is rooted in the software architecture perspective adopted by this work and the additonal reasons given below. For a (obviously biased) discussion of thick vs. thin see e.g. [Si06], chapter 10.

reasons for this focus
... dictated by necessity

There are a number of additional facts that make this emphasis seem worthwhile. First and foremost, the currently functionally most capable and complex ETC deployment in operation is the German *Toll Collect* [TC07] system. In 2007, over 610,000 thick GPS/GSM OBUs managed 12,000 km of motorway sections, levying 3.3 billion Euros (cmp. [BMVBS08]). While still being expensive with approximately 20 % costs of operations, output and performance make it a success. Improving operational efficiency – in the process lowering costs – is thus a very rewarding endeavour.

... economical adaptability and basis for interoperability

Requiring very little costly roadside infrastructure, the operator can adapt substantial parts of a GNSS/CN solution's functionality via software updates. This holds especially true for the issue of interoperability (also cmp. 3.1.3.3), which becomes increasingly important with each European neighboring nation introducing ETC: compliance of software interfaces and business logics is either achievable mutually on a common technological basis, or unilaterally with a flexible, capable platform. So, a GNSS/CN implementation might interoperate with a DSRC system, but not conversely. If the respective update processes are stable and handled reliably, they are significantly more economical than construction measures or large-scale hardware modifications.

... future standards

On the European level, we find another reason for our focus in the *European Interoperability Directive* [EU04], [EU042]. Its article 2 (technological solutions) states a clear preference for GNSS/CN ETC solutions, arguing for future-proofness and versatility. Also, the directive explicitly mentions value added safety and information services in the context of ETC. While especially services with central provisioning (cmp. 2.1.2.1) may be supported by thin clients, thick clients arguably represent a more flexible platform for VAS, as they can provide more run-time resources for sophisticated additional processes. For the German tolling system, [EU04] is reflected in [BMJ05].

A more detailed account of OBU component requirements is given in chapter 3 (technical specification) of [RCI07]. The EC-funded *Road Charging Interoperability* (RCI) project aims at establishing a technical basis for a *European Electronic Toll System* (EETS, [EU09]). As minimal OBE connections, it requires both GNSS/CN and DSRC interfaces. The explicitly stated compatibility to the German system implies a thick client. For the work given here, this line of argumentation leads to a tendency toward a thick platform architecture in order to enable support for a wide range of potential ETC realizations, anticipating their specific requirements.

relevant patents, general structure

For the overall structure of a GNSS/CN ETC system with thick client OBE, we have to follow [DC021], [DC022] and the related sketch of [DC06] regarding general composition and functionality. These patents have to be considered the currently relevant that come closest to the approach on tolling presumed by this work. However, while the general ideas seem valid, the system illustrated by [DC021] and [DC022] is too incoherent, informal and unfocused for our purpose, coming from an engineering point of view. It does not provide a distinctive software architecture, intermingles static and dynamic aspects, and unnecessarily couples software with hardware. Furthermore, partially (e.g. concerning the tariffing algorithm), it is only valid for a specific tariffing solution, of the type implemented in Germany. Anyway, we can identify the elements as listed in table 2.3.

our view on an ETC deployment

To better suit the purpose of this work, and still accepting the general concept of [DC021] and [DC022], we have to map the components to a deployment for future reference, in the process complementing the structure with omitted but relevant elements. The result is illustrated in figure 2.6.

In accordance with our basic configuration of a telematics infrastructure from section 2.1.2.1, we discriminate distributed fixed, mobile and center-side components. Central communications and gateways are implied.

the center side

We further differentiate the central processes of [DC021] into *commercial administration*, *customer relationship management* and *basic/operating data processing*. From our experience, the software of the former two components is very different from the latter. For the commercial and CRM domains, established standard solutions are widely known and available, e.g. SAP. They have to be customized for the ETC system interfaces and domain, but that is also industrial standard procedure. Correspondingly, the operational risks and potential problems are generally manageable and known.

Basic and operating data processing on the other hand currently are still for the larger part proprietary implementations. This is due to the fact that while there are a number of standard map solution providers – with the associated map editing applications –, ETC demands a guaranteed accuracy and specific attribution of geo data. Additionally, details of the locating algorithms may be

Components	General administrative *central processes*, a *monitoring/control system*, a *GNSS*, a *service point* and the *user*. An *automatic toll collection system* consisting of an *on board unit*, *OBU service center*, *charging data management* and *support beacon*. The OBU is structured in three layers: The *ETC application* consists of the modules identification algorithm (for road segments), tariffing, communications process, user interface and control process. An *operating system* provides priority management, manipulation detection, logging, version control and access to the hardware. The *OBU hardware* consists of modules for communications (GSM and DSRC), navigation (GPS and additional sensors) and a control unit with HMI, processor, memory, a smart card and service interfaces.
Functions	As primary functions of the automatic toll collection system, *road segment identification*, *charging*, *transmission of data*, *establishing access to the ETC system*, *supporting control/enforcement* and *warranting and monitoring correct charging* are defined.
Interfaces, Data Flow	The central processes define and communicate *OBU operating data sets* (toll road and tariff data) to the OBU service center and service points for versioned distribution to OBUs in the field. GNSS and support beacons provide *positional data* to the OBU. The OBUs communicate *charging data* to the charging data management. *Receipts* and *status information* are transmitted to monitoring and central processes.

Table 2.3: Digest of a GNSS/CN ETC system as introduced in Germany and extracted from [DC021]

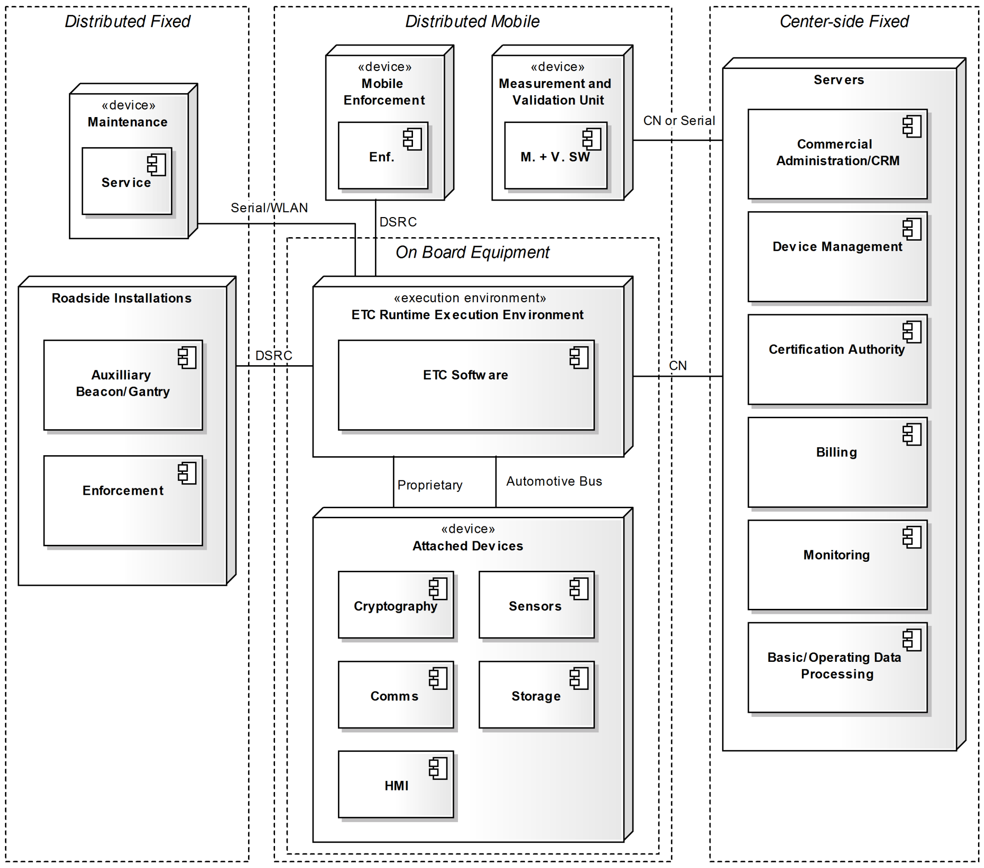


Figure 2.6: ETC Deployment Overview

classified, including their operating data specifications. For the time being, the respective processing is handled by the ETC operators to minimize both costs involved and the risks of faulty operating data entering the charging process. Thus, this distinction is relevant for our hazard analysis following in chapter 3.1.1.

To extend its scope and stress the importance, we generalize the OBU service center to a universal *device management*. This still encompasses the systematic provisioning of OBE with operating data. Additionally, experience has shown us that maintaining the OBE in workshops or garages (as suggested in [DC021]) is very costly due to the necessary service point network and physical presence of the vehicle. While this is unavoidable for the hardware equipment, the operator may update many parameters and modules of the ETC software remotely, over the air (OTA). This is common practice in the domain of mobile telecommunications (cmp. e.g. products based on [ETSI96]). For ETC, it significantly improves economy of operations, and thus is relevant for this work.

A *certification authority* (CA) manages the center-side of system security: administration of keys, certificates and the OBE SAMs (see below). Due to the high criticality of ETC security, commonly the operator establishes a dedicated CA.

Billing on first glance may seem a standard component similar to the commercial/CRM elements. Depending on the respective toll atoms and tariff scheme of the actual ETC implementation however, mapping to the CDRs of a standard billing system may or may not be trivial.

Monitoring realizes a number of functions on different levels. It is used to request and receive status information of OBE, process the results of distributed enforcement components for fraud detection and provide high level operating figures on system performance, e.g. load, transactions per time unit and overall failure rate.

the distributed side

A direct result of the necessity for in-house basic data compilation (see above) is the *measurement and validation unit*, responsible for gathering raw data on the toll atom geometries and validating the resulting OBE operating data sets with actual road profiles.

Mobile enforcement components enable authorized personnel to access the OBE software of a vehicle and compare the stored parameters and charging data with observed details. Fixed roadside-installed enforcement components feature sensors to automatically take measurements of vehicles for comparison. Conspicuous data is forwarded to monitoring.

Even with OTA device management (see above) handling the bulk of OBE maintenance, a certain number of service points are needed. In the cases of the analysis software reporting hardware failures and subsequent inaccessibility of the OBE software, physical replacements are indicated.

Unlike [DC021], we regard the GNSS itself as completely out of the influenceable scope of the ETC deployment. Consequently, it merely appears as one of the sensor components providing some interface to positioning. Its output may be evaluated for quality and even enhanced in certain ways (e.g. sensor fusion algorithms), but the ETC software has to accept any GNSS solution available. The one type of ETC element offering a position update as defined by the operator is the *auxiliary beacon*. In a GNSS/CN system, these are installed in places where GNSS is not sufficiently reliable concerning selectivity: urban environments, close roads with similar geometry. In these scenarios, the beacon signal overrides and corrects the GNSS position.

In the context of this work, *on board equipment* (OBE) designates a general, automotive platform for ETC software. It encompasses, but is not limited to the dedicated OBU that we currently find e.g. in Germany. That means especially that the OBE is not necessarily integrated in one device. It comprises

- a *run-time environment* (or run-time execution environment; RTE) able to process the ETC software with sufficient computing power, e.g. concerning real-time constraints, RAM, ROM (for code and configuration) and I/O interfaces,

- a *static storage* to hold an operating database which has to be able to cope with the automotive-specific incidents like sudden loss of voltage without losing integrity,
- a *cryptography* module, i.e. secure access module (SAM), encapsulating cryptographic functions, keys and certificates, commonly a smart card,
- long and short range *communications* interfaces for message exchange over CN and DSRC modems,
- a GNSS receiver and additional *sensors*, e.g. odometer, gyroscopes, to support and enhance the GNSS locating algorithms and
- a *human machine interface* (HMI) for interactions with the driver; manual tolling scheme parameter declaration, visual and/or audible output.

The RTE and the devices may be connected by some proprietary interfaces, including single-mainboard integration, and/or standard automotive busses as previously introduced. Note that we imply some wrapper or API software for all components and interfaces, i.e. the nodes of the deployment diagram strictly represent hardware, the components software.

2.3 Aspects and Business Processes

After establishing the general structure and functionality of an ETC system, we can now narrow the focus on the ETC OBE software. While the previous section defines a number of assumptions, substantiating a GNSS/CN thick client approach, we still have the need for an abstract view on the ETC application, consciously avoiding unnecessary restrictions for an actual solution, e.g. by directly describing concrete components or functions in an early stage of conception. On the other hand, this abstraction should be valid for any ETC OBE realization, not restricted to the approach preferred here.

the question for ETC software characteristics

Today, we find a multitude of ETC concepts and installations. Business logics and technical components of the British city access pricing of London, the Swiss undiscriminated tolling of kilometers, Austrian and German dedicated motorway tolling – some elements they share, some are completely unrelated to the other. As they still belong to one family of systems, the question for their essential aspects and common characteristics arises: what defines an ETC system from a software point of view?

the notion of aspects

The desired high level of abstraction is achieved by establishing the ETC-specific core *aspects*. We take the notion of aspects from [FECA04], denoting *cross-cutting concerns and properties* that are not necessarily congruent with some specific module, class, function or other decomposed element of the respective language (design or programming). They may also encompass non-functional characteristics of the software. Instead, an aspect can pervade a number of system components, or, vice versa, a set of elements may implement an aspect. As we will see below, an ETC software features very tightly integrated interdependencies concerning its processes and component interactions, e.g. with the omnipresent security. This may result in situations where a decomposition is necessary to manifest a property, but not feasible, or at least not sufficiently comprehensible, if based on classes etc.

On a side note, [BM98] and [MNS95] give a related, very pragmatic and practical solution in the form of *conceptual modules* and the *software reflexion model*. Conceptual module querying is typically used for reengineering purposes of source code and potentially for design conformance checking. Only based on line numbers, variables and procedures, it allows to establish a relation between a set of these and a logical unit – the conceptual module, e.g. some "aspect" – very rapidly.

reasons for the aspect view

Aside from the software design, identification and generic description of these aspects has pragmatic reasons: as the software defines the functional behavior and implements the intended business processes of the system, it determines many components of the costly hardware infrastructure. By systematically examining the actual requirements of a tender invitation corresponding to the core ETC aspects, potential platforms may be evaluated transparently and purposefully (similar to [Ste03]). Furthermore, enabling systematic feature classification allows better comparison of ETC systems, helping the pursuit of interoperability by highlighting the factors to harmonize. Finally, a traceable implementation of explicit aspects supports the selective advancement of specified features by improving only the associated elements. Vice versa, changes to the platform (e.g. due to general technical advancements, or supplier changes) may be examined for functional consequences. Aspects may be mapped to a deployment and reflect themselves as conceptual modules or cross-cutting concerns in the implementation.

the ETC core aspects

Figure 2.7 illustrates the core, primary aspects of an ETC OBE software system as identified by this work:

locating – geometric on the physical traffic infrastructure and logical on the corresponding tariff scheme graph, i.e. the decision if the vehicle is riding a toll atom (road segment, area, etc.) and which (identification),

charging and payment – determining the amount due and processing the transaction to produce a receipt,

communications – exchanging structured information between distributed components of the system,

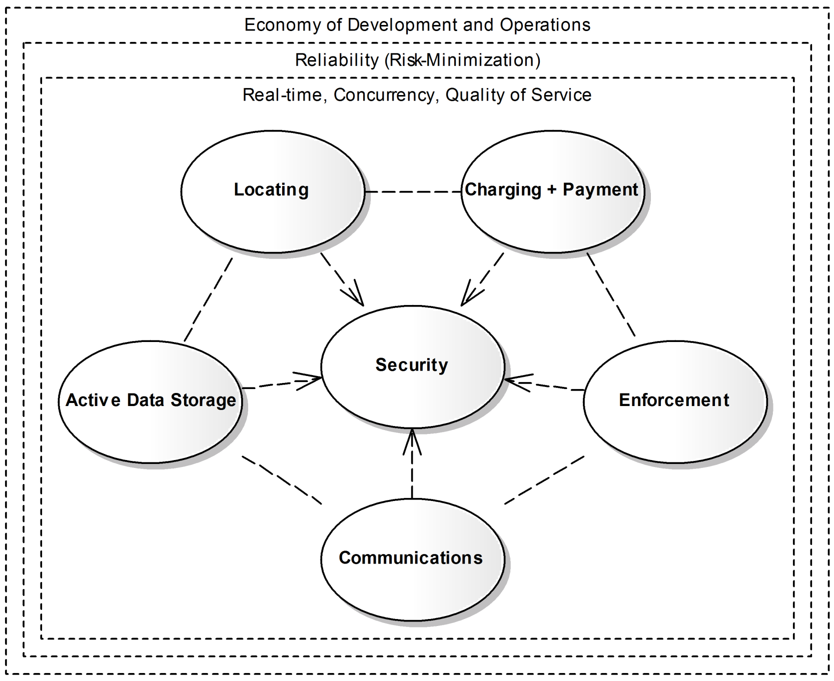

Economy of Development and Operations
Reliability (Risk-Minimization)
Real-time, Concurrency, Quality of Service
Locating
Charging + Payment
Security
Active Data Storage
Enforcement
Communications

Figure 2.7: Core ETC Aspects

active data storage – memory/access optimized storing and selective, expiry-date-dependent synchronization of operating data,

enforcement – checking the consistency of traffic infrastructure access and tolling transactions, identification of trespassers and

security – cryptographic measures to ensure privacy, authenticity and integrity of the system's processes and their data in a

concurrent, real-time context – time is a critical processing parameter for a set of activities that need to be executed simultaneously in order to fulfill all quality of service requirements the software is confronted with.

Consequently, we regard associations to billing, subscription, CRM, etc. as secondary and outside of our focus. Interdependencies between the aspects induce pre- and postconditions for their respective activities, e.g. accessible resources and yielded results. An invariant of all aspects is operational security, i.e. functions for en-/decryption and signature are available with current interfaces, keys are present and valid. Failure postconditions and resulting problems are subject to section 3.1.1.

2.3.1 Locating

Preconditions: means of positioning (services, devices) operational, map data current and accessible.

positioning, operator matching, toll atom matching

The aspect of ETC locating is manifold, with three ordered refinements. First of all, it implies the positioning, i.e. getting a fix on the vehicle's physical location. In a GNSS system, this may typically be obtained from a GPS receiver in the form of latitude, longitude, heading, speed and estimation of deviation data. Second, a rough geographic matching of the position to an ETC operator's managed area, i.e. a polygon encompassing toll traffic infrastructure. Note that this matching is not reduced merely to a *true* (inside area) or *false* (outside area) decision, but also has to detect the approach toward an area, notifying the active data storage (see below). The third refinement matches the vehicle's position to the geometries of a toll atom, e.g. the segments of a motorway. Given a DSRC ETC solution, the aspect of locating is still valid, but the refinement order may be reversed. By receiving a logical ID from a beacon, we can deduce the vehicle's physical location.

logical view

Besides the geographic respectively geometric view, locating refinements two and three also include a logical view, yielding the result essential for ETC: selecting the currently active operator ID to apply the corresponding business processes and tariff scheme, and identifying the toll atom ID as reference for the tariff scheme data structures and pricing algorithms.

Postconditions: valid operator ID, current toll atom ID or outside operator area/toll atom.

2.3.2 Charging and Payment

Preconditions: relevant parameters including toll atom ID declared and current, tariff scheme current and queryable, communications operational.

commercial transactions

Charging and payment encompasses all structures and mechanisms to implement the commercial transactions, starting with a toll atom to process. A tariff scheme, provided by the active data storage, is queried based on the current vehicle, driver, time and operator parameters and applied to the atom (rating and pricing), resulting in an amount of money. On that, the application charges some user account and generates a receipt, finalizing the transaction in a pre-paid system. In the case of the thick client, this is handled on-board, i.e. the signed receipt represents a cash-equivalent entity that should be protected against manipulation or loss.

receipt handling

From the view of the OBE software, the toll atom is *cleared* with the generation and signing of the receipt. Consequently, it can be presented to an enforcement as proof of correct and accepted charging. Eventually, on reaching a defined limit in a post-paid system, the OBE transmits the receipts to the billing servers via communications (see below), where the previously charged account is balanced. The respective transaction interfaces (e.g. receipt formats) and protocols are also associated with the aspect of charging and payment.

Postconditions: corresponding account(s) charged or balanced, last/current toll atom cleared, receipt/proof available for transmission to center or enforcement, or transmitted/validated.

2.3.3 Active Data Storage

Preconditions: file/memory resources operational, locating and communications available, update triggered externally (optional).

operating data

The ETC OBE data storage provides the operating data (OD) to the application's processes. Operating data management concerns received data from central servers as well as persistent structures generated and updated by the OBE software itself. It consists of

map data – geographical references of the area managed by a specific operator, geometrical descriptions of toll atoms (virtual toll points, vectors, areas etc.), optionally surrounding traffic infrastructure to enhance differentiation between toll and toll-free sections,

tariff scheme – an update- and queryable representation of the scheme introduced in 2.2.1,

configuration – the set of currently valid system parameters of the OBE hardware and software components, access information for the interaction with ETC operators (MSISDN, protocols),

set of receipts – an ordered collection of cleared toll atoms. The order may be defined e.g. by date/time, operator, location or tour.

database realizations

Persistent storage, updates and querying of map and tariff data imply some kind of database. This may be implemented as single table files for simple data structures, a proprietary embedded database, any standard ISAM or SQL solution, depending e.g. on the degree of efficiency required on the hardware platform due to performance or memory restrictions. Receipt datasets may be stored as single files in a directory, or as list in a file depending on their complexity and order. Storage of receipt data also includes memory located on a smart card.

The aspect of data storage in an ETC environment has to implement specific characteristics regarding active updating and integrity.

active data management, OD invariant

Active data storage in this context implies validity time intervals and expiration dates for externally provided operating data release versions. The OBE data storage has to recurrently check all of these objects and actively initiate updates if the data is about to expire. A similar mechanism checks sum limits of receipt sets and initiates transmission to billing. In the process, the invariant condition of current and valid operating data must never be violated for the application as a whole.

Consequently, update management and scheduling of the storage has to consider the fact that it is not guaranteed that the OBE may access the corresponding operator's device management center via CN at any given time. Besides the time constraints, updates are thus additionally determined by location: if the need arises, the active data storage has to detect the approach of an operator's area to enable the timely update of expired data that was not accessible before. A final update scenario is triggered by an external cue broadcast by the center in the case of an unscheduled modification due to abrupt and temporary alterations to traffic infrastructure e.g. induced by road damage.

The above invariant also requires that the incoming operating data structures are completely and correctly stored on the OBE before becoming valid. With the expiration of the previous set, they are activated, the obsolete set discarded. As our software is potentially operating in a resource-restricted embedded environment, data management and storage has to be optimized for memory usage, as it has to handle more than one full operating data set version at a given time. This can be achieved e.g. by selective updates of specific elements only, or "delta" versions of operating data sets, describing the differences between two releases.

ETC data integrity

Database integrity constraints are also tightened for the ETC use case. Commonly, coherency and consistency in data structures may be achieved by the introduction of transactions: insert, update or delete operations on data records are explicitly commited; if they fail, the structures roll back to the state before these operations. Due to the rigid time constraints for automotive system shutdown (cmp. sections 2.3.8.4 and 3.1.4), the software has to ensure that no data loss may occur in the case that the software shuts down before a record is commited from local variable/object/memory to persistent storage. This is especially critical in the case of signed receipts stored on a smart card, demanding either sufficiently efficient transaction implementations or the general safe, static storage of these objects.

Postconditions: operating data current and queryable.

2.3.4 Communications

Preconditions: communication devices operational, connection configuration parameters, keys and certificates current and valid.

basic management

The OBE software has to manage the different long and short range transport media with their specific characteristics as stated in 2.1.2.2 regarding bandwidth, time constraints and availability. This management – corresponding to the ISO OSI [Zi80] physical, data link and network layers 1 to 3 definitions – starts with maintaining a status of operativeness: continuous responsiveness of the respective device, signal strength where available, registered coverage of configured CN operators. A basic level of communication protocol defines how to establish, maintain and release a connection including reacting on interruptions in the case of connection-oriented media.

incoming signals

An important distinction lies between outgoing and – if permitted – non-OBE-initiated incoming calls, as the devices have to be polled for received signals (it is not sensible to generally presume an interrupt mechanism of the HW platform). Here, the software has to find an optimum polling frequency between excessive workload (busy waiting), message queue overflow and missing a real-time deadline.

timeout handling

Concerning time constraints, both long and short range transaction protocols from OSI layer 4 (transport) upwards need to regard timeouts, albeit of different magnitude. Without prematurely touching the application layer 7, messages sent over CN generally require an acknowledgement to ensure correct reception and the completion of a transaction. This holds true for both directions, e.g. the OBE confirming a new version of operating data, or the center confirming a set of receipts. In the case of timeouts, the information may be retransmitted after a defined time, e.g. seconds or minutes in the case of a dead spot, or dependent on location, e.g. on reentering an operator's area after the OBE left it an indeterminate time before (note the interdependencies to locating and active data storage above).

short range time constraints

For short range transactions between passing OBE and roadside installations, we find strict deadlines for completion. These can be comparably short, e.g. if we have to assume a maximum speed of 250 km/h on German motorways. Applied to the example media performance parameters from 2.1.2.2, this leaves little time/space for complex protocols or more than a few bytes of user data. Consequently, a timeout in the worst case equals to a failed transaction for the short range communications scenarios.

message formats and exchange protocols

Mechanisms for the handling of message exchange between OBE and center are also directly associated with the aspect of communications. Interdependent with the security aspect (see below), this implies the mutual authentication, en-/decryption and integrity safeguarding of the messages. The previously stated connection parameters bandwidth and timeout constraints determine the message formats and protocols:

- binary encoding of single message overhead and user data, defined as e.g. *TLV*, *ASN.1* with *Packed Encoding Rules* (PER) to optimize memory consumption, cmp. [ITU021], [ITU022],
- management of message sequences; segmenting user data in a number of chained messages in the case of a connectionless medium like SMS, order of message exchange, e.g. *challenge/response*, *ACK/NAK*, reacting on errors, aborts.

component interface

On the boundary between the communication components and other software modules, the aspect relates to the handling of application objects to communicate from and to the OBE, primarily the serialization and deserialization from record/object to and from one or more binary arrays. This level may also provide a uniform interface for all types of communications in the form of virtual connections for connectionless media, e.g. by implementing some *open* and *close* for a message transmission via SMS.

Postconditions: objects successfully transmitted or received objects ready for processing by other application modules.

2.3.5 Enforcement

Preconditions: toll atom clearance receipt current and available, communications operational, enforcement challenge received.

extending charging and payment

While – as was established in 2.2.1 – seemingly secondary to the processing of tolling itself, and actually only indirectly maintainable in ETC systems other than thick client solutions with real-time charging, the aspect of enforcement irregularly extends charging and payment. This commonly can be considered sufficient to encourage participation in the tolling system. Enforcement is thus crucial to the overall ETC use case and is considered an aspect in itself. It is closely interdependent with security (mutual authentication of both OBE and enforcer, integrity of exchanged data) and communications (transmission medium DSRC).

OBE view, reactive feature

From the OBE perspective, enforcement is reactive, i.e. the software invokes the corresponding processes only on being challenged: the OBE communications receive a signal requesting the receipt of a defined toll atom. Additionally, the enforcement unit may readout a set of parameters determining the amount to charge and declared in the OBE software for confirmation with the actual vehicle attributes. The requested objects are selected and handed over to communications for transmission to the enforcer. With respect to a defined time constraint, the answer is accepted by the OBE.

enforcement positions and timing

Another notable interdependency can in specific cases be found with locating. The ETC operator has to ensure that fixed enforcement installations as well as mobile enforcers are positioned in a way that grants the OBE locating sufficient time to safely produce a receipt, i.e. determine the toll atom and process charging and payment. For the OBE software, the enforcement consequently implies a time constraint for the processing of toll road segments equal to or larger than a minimum length. *Example*: this constraint may assume a maximum speed of *250* km/h and a minimum length of *1* km, with a viable site for enforcement equipment after *500* m, resulting in a maximum locating, charging and payment activity duration of *7.2* sec after entering the toll segment.

Postconditions: OBE response and clearance of toll atom accepted.

2.3.6 Security

Preconditions: keys, certificates valid and available.

The ETC domain handles business-critical information; large sums of money as well as user-related data, e.g. the implied position of the person of a driver, in a distributed, not trustworthy environment. A system's operator has to account for non-repudiation and attestability of the tolling transactions. Thus, security is a mandatory, integral part of the ETC business processes both on the micro level of the OBE software components – integrity checks, access control, manipulation detection, providing privacy – and macro level of the system – OTA transactions between distributed and center elements. Security measures are commonly classified in three categories (cmp. [Ste03]).

privacy

Privacy functions ensure that information processed and communicated between OBE and central systems is not readable by a third party. This can be realized by encrypting the exchanged data, e.g. with public key algorithms. As GSM transmissions usually are already encrypted by the GSM operator, an ETC service provider has to evaluate if this would be sufficient (i.e. it just has to be made sure that a message can not be easily intercepted and read – while the GSM operator is not regarded as a possible attacker) or if the service requires an independent security module under complete control of the service provider itself.

authenticity

Authenticity functions have to validate the identity of the communicating parties. Central systems must be able to safely assume that they transact with a valid mobile unit and vice versa. By adding a unique signature to the exchanged messages (respectively over their content), a sender can prove its identity to the receiver.

integrity

Integrity functions detect manipulation – for example of data by calculating a checksum or hash value over newly generated and updated data. If on a later occasion (e.g. when receiving the data) this unique value is recalculated and differs from the previous regularly calculated, the contents have been altered in the meantime and an integrity failure has been detected. An ETC application may utilize this in the context of message exchange as well as protection of the mobile devices: Unlike the server systems, which usually can only be attacked over defined public interfaces, the mobile units are fully accessible – hardware as well as software – because of their deployment "on the road". Consequently, the integrity of the operating data stored on the devices as well as the device's physical components are integrity-checked to detect manipulation and be able to react accordingly (e.g. by shutting down the application).

SAM

While security measures permeate many components of the software, implementation of the core functionality – *encrypt/decrypt*, *hash/sign*, *authenticate* – is currently often associated with a *secure access module* (SAM), commonly represented by a smart card. It provides a sufficiently secure environment for cryptographic algorithms, certificates and (private) keys (cmp. [RE08], chapter 16).

The given pre- and postconditions notwithstanding, the aspect also spans heuristics to detect manipulation attempts: certain conspicuous usage, signal or event patterns hinting at violations, e.g. only road segments with fixed enforcement installations were charged, otherwise the OBE was turned off.

Postconditions: processing of encryption, decryption or signature check completed without violations.

2.3.7 Concurrency and Real-time

Preconditions: multiple threads of execution access resources (e.g. active data storage and payment use communications), time constraints defined.

the issue of concurrency

The preconditions and assumptions stated in 2.2.3 imply that we have to accept concurrency in the ETC OBE software, as the thick client solution features a set of invariants that need its activities to run simultaneously (cmp. next section for an explicit illustration). Due to the adjunctive interleaving of states of the software's different component threads of execution, the number of overall system states grows exponentially with the number components. The complexity of the system state configuration rises to a "state explosion" (cmp. [Ro98]), impeding controllability of the OBE processes. *Nondeterminism* of the application's behavior, *deadlocks* and *lifelocks* of threads are the potential problems arising in this context.

processes and threads/tasks

The concurrency aspect of this work differentiates *processes* and *tasks* respectively *threads*. A process generally represents a complete application like the ETC OBE software, with its dedicated memory address space, run-time, input and output resources, running in some execution environment like introduced in 2.2.3. Tasks or threads each implement sequential operations in the run-time context of a process, and access its memory and resources. They may run in parallel[7] – that means that their corresponding state traces are interleaved to produce the composite process state configuration. *Note*: this work also uses the widely established term *business process* to designate the comprehensive procedures implemented by the ETC system as a whole and its components, including processing, program and information flow. Business processes realize the system's use cases.

ETC real-time

Closely associated with the parallel processing of tasks is the aspect of real-time (RT) behavior. Actually, a number of real-time constraints demand concurrent tasking, e.g. an operating data update must not delay a tolling in process. Real-time in the context of ETC differs from the interpretation of other domains. Partially, e.g. because the timescales are different from other RT applications like automotive (cmp. RT term and scale in 2.1.1.1 to table 2.4), we find tendencies that do not regard ETC as real-time at all. This however, would be wrong, as the primary function of toll collection clearly depends on the timely execution of various subfunctions. Depending on the

[7]We use the terms *simultaneous* and *parallel* also for task-switched quasi-parallel software in the case of a single processor architecture.

Timescale	**Constraint Context**
< 1 sec	Sensor readout information availability for locating. OBE internal component interaction, en-/decoding.
Seconds	Locating, charging, receipt generation. Enforcement response, message exchange with road-side equipment.
Minutes	Transactions between OBE and center.
Weeks – Months	Operating data update time slot.

Table 2.4: Scales of ETC time constraints

viewpoint[8], ETC should be considered *soft* RT (cmp. [Do04], 1.2.1): as we will elaborate on in detail in 3.1.1, an ETC system has to fulfill an overall quality of service measured by a failure rate, i.e. here we find a statistic measurement that implies a strictly defined but nevertheless existent tolerance for faults.

RT and post-processing

A specific influence on ETC RT is the interdependency between enforcement and locating (see above). A navigation system for example is allowed to switch (observable as a "jumping" position) between location alternatives depending on the current state of the map matching algorithm, as it is expected to provide a real-time "best guess" to the user. In contrast, ETC locating has to reach a stable solution as a precondition for valid charging. Combined with the maximum speed-induced deadline for a response to enforcement, ETC locating needs to converge on a sufficiently high probability while adhering to a time constraint. Up to that point, there is no tight cohesion between current location result and time, permitting post-processing of locating data inside of the deadline time interval boundaries.

handling concurrency

To effectively handle concurrency, the software has to implement

task management to instantiate the threads, associate and provide them with run-time resources (memory, access to other threads and devices),

priority-based scheduling to systematically and fairly allocate run-time to the threads (single processor case),

inter-task communications in the form of queues and communicable messages, signals, objects, and

mechanisms to guarantee mutual exclusion on entering critical program sections, e.g. semaphores or monitors (cmp. [Be06], chapters 6 and 7).

Also explicitly associated with the aspect of concurrency are the thread priority settings, e.g. in the context of a static priority pattern (cmp. [Do03], 5.9),

[8]In 2.1.1.1, we gave an alternative reference on hard and soft RT.

to facilitate predictability by additionally analyzing task periodicity, worst-case execution times and deadlines.

Postconditions: collision free resource access, time constraints unviolated or violation identified[9].

2.3.8 ETC OBE Software Business Logics

In section 2.2.3 we introduced the deployment and components of an ETC system with a description of their distinctive purposes. Together with the previously defined aspects that detailed the primary assignments and characteristics of the OBE component software, we are now able to sketch a generic business process of the ETC OBE application. This illustrates the activities required to implement a thick client automatic tolling device – their basic sequence, concurrency and constraints. As that, they represent a section of the overall ETC system's business process, namely the distributed measuring and charging of toll atoms. The process section directly integrates with the business logics of the center and other distributed system components. While their processes are outside the focus of this work, we still regard the interfaces between them and the OBE logics, i.e. how they interact with the OBE software activities.

system state configuration; validity, consistency

An important definition in the context of the OBE software is the *system state configuration*. For any time t during the software's operation it designates the current step of execution[10], variable values and queue contents of each thread. As we will see, some requirements demand the explicit handling of this state configuration. In order to be expedient, the configuration needs to be *valid* and *consistent*, thus two corresponding invariants. Validity implies a time-, location- and state-trace-related coherence between the system state of t_i and t_{i+1}, i.e. there are no unexplained[11] clock or position offsets and no illegal state sequences in each task. This is especially relevant if an OBE shutdown lies between t_i and t_{i+1}, e.g. if the vehicle resumes a tour from the same location. Consistency requires the adherence of concurrent tasks to defined safety conditions (cmp. [Ste032], 4.1.2, [Ro98], 1.3.3). The software has to ensure that contradictory, mutually exclusive task states do not occur coinstantaneously in t, e.g. charging a toll atom while travelling a toll free area. Additionally, the data objects processed in the current state of each thread have to correspond over the system state configuration, i.e. their OD versions, timestamps, exchanged message types.

signals, events, activities, actions

Figure 2.8 presents an overview of the business logics to realize by an ETC OBE software. In a way, they also reflect the lifecycle of the ETC OBE

[9]In an integration scenario, this might be a valuable result that can hint at insufficient hardware capabilities for timely processing.

[10]Not necessarily the program counter, but a proprietary reference, as described in later chapters.

[11]Transporting the vehicle by ferry is an example for inducing an explained offset.

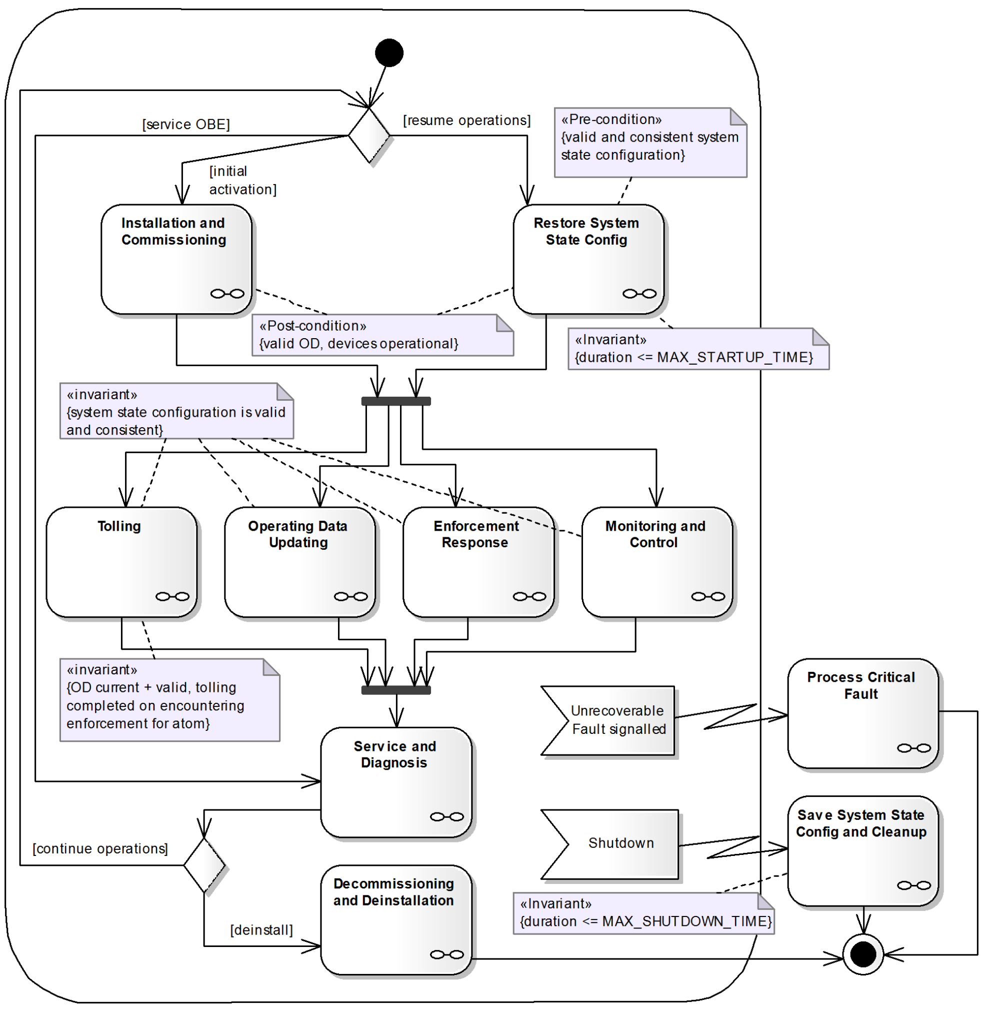

[service OBE]
[resume operations]
«Pre-condition» {valid and consistent system state configuration}
[initial activation]
Installation and Commissioning
Restore System State Config
«Post-condition» {valid OD, devices operational}
«Invariant» {duration <= MAX_STARTUP_TIME}
«invariant» {system state configuration is valid and consistent}
Tolling
Operating Data Updating
Enforcement Response
Monitoring and Control
«invariant» {OD current + valid, tolling completed on encountering enforcement for atom}
Unrecoverable Fault signalled
Process Critical Fault
Service and Diagnosis
Shutdown
Save System State Config and Cleanup
[continue operations]
Decommissioning and Deinstallation
«Invariant» {duration <= MAX_SHUTDOWN_TIME}
[deinstall]

Figure 2.8: ETC OBE Software Business Logics Overview

from commissioning to deinstallation. For the activity diagram semantics we generalize the above mentioned interactions with other system components over the given interfaces to *signal* (cmp. [OMG092], 13.3.24) input/output; sensor data, GSM messages, GPS positions, hardware interrupts. Also, we use the term *event* (cmp. [OMG092], 13.3.13) for some measureable incident in the state configuration; clock ticks, variable value changes, signal reception. Each *activity* describes the command sequence and data flow to realize a part of the system's use cases, corresponding to the *CompleteStructuredActivities* of [OMG092], chapter 12. Note that activities, or sequences of, do not necessarily equal tasks or threads of an RTE. A task may implement a part of an activity, one activity or n activities. Finally, the command sequences are assumed to be made up of *actions* (cmp. [OMG092], 11), atomic operations on the state configuration.

2.3.8.1 Pre-Operations

installation and commissioning

Physical installation of the device(s) in the vehicle entails installation and commissioning of the software. A linear activity, commonly realized by the OBE in concert with some service terminal or OTA service center, imports the application and if need be some base software or OS, followed by an initial set of operating data. The OD includes a valid initialization file with device parameters and operator contact data (server MSISDNs, IP addresses etc.). The local process is completed with the personalization and activation of the OBE. We assume a smart card-based personalization, e.g. user/vehicle data is stored on an operator-issued card that is inserted into the OBE and activated by a PIN. After contacting and being acknowledged by the center (e.g. CRM), the OBE is effectively conscribed to the operator, thus introduced and enabled to participate in the ETC system.

resuming operations

If the software resumes operations after a shutdown, it has to restore the complete system state configuration. Due to the criticality of the specific checks (see above), the activity should be handled by the ETC software itself and not left to an operating system or other third-party RTE. On successful task state restoration including data, each activity of the application is continued. The restoration activity itself is submitted to a timing constraint: its duration is limited by a maximum acceptable startup time (MAX_STARTUP_TIME), after which the driver can presume that the ETC OBE is operational and is allowed to proceed with the tour. If the deadline is missed or a fault occurs during the system state reconstruction, a malfunction is flagged.

2.3.8.2 Operations

reasons for concurrent activities

Both the commissioning and continuation include a general checking of runtime resource and device availability, so that after successfully completing either activity, proper OBE software operations can begin. With the evolution from sequential to concurrent sections, these present an important trait. In pre-operations, installation as well as restoration after shutdown consist of a sequence of steps. During regular operations however, i.e. tolling, system use cases and invariants dictate parallel, synchronous activities due to the following reasons:

1. *Invariant "active tolling" and invariant "OD current"* – From the overall ETC application domain it should have become clear that it is imperative that the operational OBE is able to toll at all times. A precondition for a valid processing of toll atoms is current operating data, in turn requiring timely updates over the air. As update scheduling is nondeterministic,

i.e. initiated by irregular events like reaching the OD expiration date or external signalization, both tolling and update activities need to run simultaneously, so that a necessary update does never preempt requisite tolling run-time.

2. *Use case "enforcement" with above invariants* – An enforcement challenge is another event the OBE software is confronted with occuring nondeterministically. The enforcement invariant itself notwithstanding, an OBE has to respond with its status immediately on contact with an enforcement instance to be considered operational. Consequently, the enforcement transaction handling runs concurrently to tolling and OD updates.

3. *Use case "monitoring" with above invariants* – Permanent monitoring and control of valid ETC processing is a critical precondition for binding monetary transactions and their attestability. Monitoring activities of the OBE impact both the micro level – the software of each distributed device – and macro level – the aggregate collaboration of all devices and the center – of the ETC system. First, the software needs to continuously log critical regular, conspicuous and fault incidents to support a directed encircling in the case of problems. On a larger scale, the center requires information about the overall operational availability and potential hazards, e.g. increased occurrence of locating or accessibility disturbances. Thus, it would not be sufficient to interpret monitoring in a passive sense, e.g. in the form of a logging function to call by the components to observe. Instead, we have to consider OBE monitoring an active task, also in the sense of an independent component: it may be used by other modules as a protocol instance, but it is also able to register and flag problems like irregular behavior/traces or blocked threads.

the four concurrent ETC activities

We find four overall activities processed simultaneously, of which three – *operating data updating*, *enforcement response* and *monitoring/control* – provide auxilliary functions, thus ensuring the application's adherence to constraints of the primary tolling activity.

operating data updating

Operating Data Updating Operating data updating manages versions of the tolling area, road segments and tariff schemes stored on the OBE. Thus, it checks and ensures the validity of current operating data. This validity encompasses a number of constraints. *Completeness* requires that all accessible toll segments are available in the data set and associated with tariff data for any accredited vehicle and date. *Temporal and geographical coherence* demands a gapless succession of operating data set versions and road segment chains, while *consistency* in this context refers to non-overlapping durations of validity, unique versions, segments and unambiguous attributes. Whenever triggered, e.g. by a received signal or a timer event, the updating activity requests status of the operating data, processes insertion of new records and reorganizes the database structures accordingly. Besides the apparent association with the aspect of an active data storage, this activity is associated with the aspects of communications and security to process transactions over the air and effect data authenticity and integrity respectively.

monitoring and control

Monitoring and Control Monitoring and control perpetually checks all other activities for fault states, warnings, errors or otherwise suspicious behavior. It maintains validity and consistency of the system state configuration. Beyond local logging of regular events and other incidents this activity may interact with a central service to transmit aggregated monitoring information and receive commands to control the configuration of the device. Command sequences received via CN (aspect of communications) from the outside are authenticated to ensure a legal source (aspect of security). Depending on the level of criticality set by general policies, outgoing system logs and notifications may be encrypted in addition to integrity-verifiable (e.g. by adding a checksum to the respective message).

enforcement response

Enforcement Response The enforcement response merely reacts on a challenge from mobile or roadside checkpoint equipment: the activity reads out the required data from the smart card and memory. However, timing of the response is crucial to prove correct operativeness and produce evidence of payment. This authenticated message, transmitted via DSRC, may carry a single toll atom charging confirmation or a tour, i.e. chain of road segment receipts, and currently declared vehicle parameters for additional plausibility checks – coherence of segments, gaps in a tour, set versus observed vehicle attributes etc.

configuration, initial location

Tolling The tolling activity itself (cmp. fig. 2.9) consists of a number of sequential steps that fork out to concurrent threads of execution. Initially, tolling relevant attributes (2.2.1) are set by manual input from the user or reading a configuration file. All run-time resources (memory, sensors, devices) are checked for availability and operation. If the software and OBE meet all prerequisites, the application determines its current location. From that, it derives the responsible operator, if a query of the operating database yields coordinates of a corresponding area that encompasses or is close to the current location.

setting active operator and account

Approaching or inside the operator area, the application activates its dataset: the tariff scheme is checked for applicability conforming to the time, date, user's and vehicle's current attributes. To process charging on-board, the software reopens an account in the case one exists that was not yet closed and transmitted to the billing systems. Otherwise, it opens a new account, if no defined limit – based on sums, time or storage – denies that. Accounts are commonly kept securely on a smart card, thus the storage limit is relevant even if the related data records are small.

toll atom detection

When a debitable account is available, recognition features, e.g. geographical references of toll atoms are then submitted to detection by GPS or DSRC. Depending on the specific locating aspect's implementation, the references may be organized in map subsets, and take the form of vectors for continuous matching or virtual toll points for discrete matching. In any case, regular position updates represent input for a toll atom detection loop. On the identification of a specific atom – a sufficient sequence of positions matches to the corresponding geometry, or a DSRC signal was received – the atom's record is attributed with relevant data: date, time of day, configured vehicle and user characteristics.

rating and charging

From the atom attribute values, the software compiles a tariff scheme query and accesses the operating database. The resulting record may directly contain the amount associated with the atom's use, or (parameters of) a function to calculate the amount. When the amount to toll is determined, the thusly rated atom is charged: the application passes ID, amount and timestamp to the smart card, which stores it in the currently open account, in effect debiting it. Additionally, the card generates a signed receipt as proof, consequently clearing the atom. This receipt may then be used in response to enforcement challenges.

At this point of execution, the tolling activity forks. It is necessary to continue immediately with the detection of toll atoms, as the vehicle is still considered as moving.

area updating

For that purpose, the working map subset resulting from a defined area surrounding the vehicle's position is updated; the process removes farther geometry and adds close toll atom recognition features. Elements of the subset can then be submitted to detection.

account management and transmission

Additionally and concurrently, account management maintains adherence to the commercial constraints for continued tolling. If the last debit operation reaches no account limit (see above), the currently active account is still debitable. The activity joins with that of the update after both are complete, tolling is resumed. In the case of a reached limit, the activity forks again. It opens a new account and again – if the operation was successful – joins with the update to continue tolling.

account balancing

The other thread of execution closes the previous account, encodes it in a transmittable message format and signs it to ensure integrity and authenticity. A scheduled[12] or directly initiated interaction with the center servers hands over the account via CN for billing, in return accepting a receipt. The receipt

[12]Note that an account is not necessarily passed to the center immediately on closure. Due to a preset schedule or data transfer volume optimization, accounts may be collected and then transmitted in packages.

is checked for integrity and authenticity, and stored on the OBE, associated with the corresponding user account, which is now considered balanced by the ETC software. This activity then terminates, with the others continuing.

2.3.8.3 Post-Operations

activities beyond operations

Either by direct service access after initialization, or by terminating the primary activities, the system reaches service and diagnosis. In contrast to monitoring and control of the operations section, this activity handles maintenance of the software in a halted state and post-mortem analysis of crashed OBE.

service

While the application may exchange or update certain software components during regular operations and still function properly – e.g. the database as mentioned above –, others, most notably hardware components like a smart card, cannot provide their required functionality during that kind of intervention. Respectively, if not redundant, tolling cannot continue with sufficient reliability during maintenance of these components.

These service and maintenance interactions with external systems located in a workshop or center-side are exchanged over a specific and secure service interface. It may be realized as either a fixed link or OTA, providing methods to modify critical components, shut down the OBE for hardware modification and configure new or updated modules on reentering this activity after reboot.

diagnosis

Diagnosis reads out system logs (states, events, messages, free memory, signal quality etc.) in the context of regular maintenance to check for suspicious entries that may hint at probable future problems, a use case similar to the above monitoring during operations. In the case of critical failures and system crashes, the logs have to support fault reproduction. Their data – leading up to the point where the software stopped logging its activities – is then recovered and analyzed post-mortem, i.e. after the OBE ceased to function.

decommissioning

After service and diagnosis, the OBE software may either continue operations, or be taken out of commission. This implies deinstallation of proprietary software components, extraction and thorough, physical deletion of data (e.g. personal-/customer-related; tours, payments etc.). Crucial hardware components are removed (e.g. smart cards with keys) and the remaining equipment is prepared for a potential refurbishment depending on condition and age.

2.3.8.4 Terminating all Activities: Shutdown and Error Handling

shutdown

Any time during processing of the given activities, the software may receive the signal to shut down. In this event, it only has a limited amount of time to orderly finish all activities, indicated by a time constraint MAX_SHUTDOWN_TIME. This is generally owed to the installation of OBE in an automotive environment e.g. potentially concerned with hazardous goods logistics (cmp. 3.1.4). It may not be required for every realization of the software, but has to be considered in the design all the same.

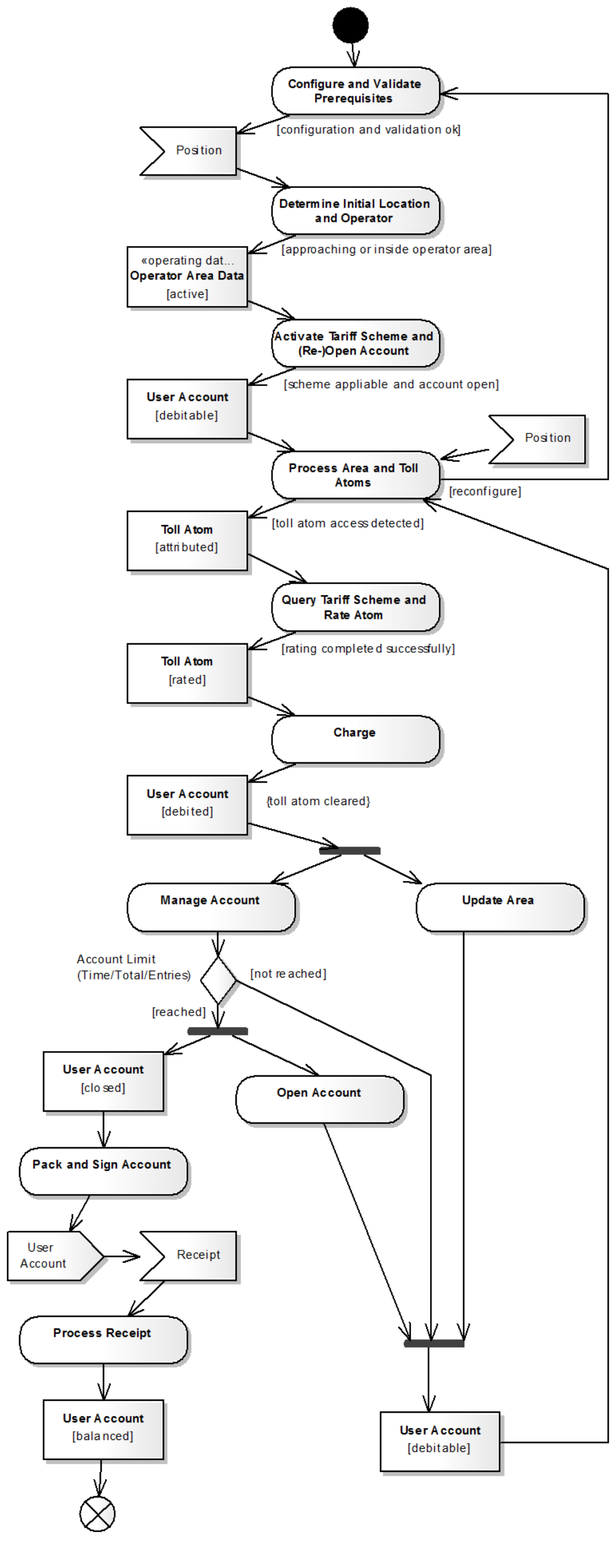

Configure and Validate Prerequisites
[configuration and validation ok]
Position
Determine Initial Location and Operator
[approaching or inside operator area]
«operating dat...
Operator Area Data
[active]
Activate Tariff Scheme and (Re-)Open Account
[scheme appliable and account open]
User Account
[debitable]
Position
Process Area and Toll Atoms
[reconfigure]
[toll atom access detected]
Toll Atom
[attributed]
Query Tariff Scheme and Rate Atom
[rating completed successfully]
Toll Atom
[rated]
Charge
User Account
[debited]
{toll atom cleared}
Manage Account
Update Area
Account Limit (Time/Total/Entries)
[not reached]
[reached]
User Account
[closed]
Open Account
Pack and Sign Account
User Account
Receipt
Process Receipt
User Account
[balanced]
User Account
[debitable]

Figure 2.9: The Tolling Activity

error handling

Handling of errors is often constricted on embedded platforms like the OBE. Resources are limited, with little redundancy, leaving little avenues to recovery. Even if a fallback or plain reboot is possible – automatically or interactively initiated by the user –, the software will potentially at least abandon a sequence of toll atoms. As we will see in detail in the next chapter, failure of a single link in the chains of process activities means failure of the application with utmost probability. Hence, the logics given in this section handle any activity's signal of a critical fault by logging and preserving as much information about the incident as possible before terminating. By these means, subsequent recovery and systematic analysis is enabled (s.a.).

2.4 Chapter Conclusion

In contrast to existing works, we introduced the domain of ETC from a strictly software point of view. Specifically, the introduction took the perspective of the OBE software, an embedded, distributed component of the deployment, which implements mission-critical processes.

conflictive parent domains

The initial sections of this chapter gave an overview over the automotive and traffic telematics software domains, both representing a parent domain of ETC with differing development approaches, as shown in 2.1.1.4 and 2.1.2.3.

Characteristical attributes of an automotive ECU application are sequential processing, deterministic interactivity with other components limited to the vehicle bus network, real-time stateless transformation algorithms with input and output consisting of signals, i.e. plain numerical values. Quality requirements are high due to safety-critical functions. Traffic telematics applications on the other hand typically rely on protocol-defined interactions with widely distributed components over various interfaces, car to car, roadside or center OTA, dynamically exchanging messages with complex structures. Concurrent processing is common, as reaction to events is triggered nondeterministically, entailing forking of execution. Requirements to reliability are generally lower, as application output is non-critical respectively informing for most use cases.

description of the ETC software domain

We depicted the domain of ETC software as a fusion of these two contrasting fields, with additional features distinguishing it from its parents and justifying its originality, e.g. a specific view on RT, interoperability and reliable management of a significant money flow. This included a general introduction to toll collection, the notion of toll atoms and their processing, expanding on tolling automation, a corresponding system deployment and focusing this work on the specific thick client OBE.

A set of aspects – locating, charging/payment, active data storage, communications, enforcement and security – manifested specific ETC functionality and features that need to be implemented by an ETC OBE software and at the same time describe it sufficiently. The chapter complemented these aspects with generic ETC OBE business logics, further substantiating the aspects, relating them to pre-/post-/operational activities and illustrating the need for concurrency and real-time behavior.

characteristics of ETC software

Based on these intermediate results, we can now explicitly acknowledge the characteristics of ETC software for the given context. And apparently, the conflicting traits of both parent domains need to be harmonized by the ETC domain, providing a challenge to system design and implementation.

Structure – The set of unique aspects realized by an ETC OBE software implies a complex structure, both concerning a set of many associated components and the data handled and processed by them. This encompasses mechanisms for reliable interactions between the active and passive elements that go beyond direct function calls.

Behavior – Business logics and events during tolling operations imply complex, state-based behavior, i.e. concurrent threads of execution reacting on the non-deterministic occurence and reception of events and highly interactive signals. All the same, processes adhere to real-time constraints. To comply with the interoperability directive, the software has facilities to manage different ETC business logics, e.g. in the form of services.

Interfaces – Interactions with the various actors beyond the OBE are diverse regarding both associated hardware and heterogenous protocols of usage. Interfaces have to meet human users (MMI), as well as handle continuous automotive sensor readouts and time- or event-triggered, i.e. irregular, OTA transactions with center servers. Flexibility in definition and implementation supports interoperability, i.e. specifications need to be open to adaption to other ETC solutions, with correspondingly customizable software modules.

Security – At the same time, the software operates in an open, insecure environment. As outside interfaces are exposed to attacks, application logics and data are protected by constant plausibility, integrity and authenticity checks.

Quality – While the ETC software domain borrows structural and behavioral complexity from telematics, reliability, robustness and overall quality of service take after automotive requirements: the processed fees induce high criticality. Due to the software's described structure and behavior, straightforward application of most automotive ECU validation methods is hardly feasible, as formal preconditions cannot be met. Thus, we take alternative approaches to ensure quality.

Platforms and RTE – ETC software has to cope with varying and evolving hardware platforms and run-time engines. The tight integration with well-known hardware – found in the automotive ECU development as well as first ETC generations – is abandoned in favor of open solutions and future European markets for OBE. Behavior of the run-time environment, e.g. concerning timing and memory management, may strongly influence the ETC software, anyway. In the event of failures, the application supplier still has to prove that its product worked correctly. The software thusly exhibits transparent and reproducible behavior.

making use of the chapter's results

What do we gain by these results toward our aim of a reliable and economic architecture of the introduced application? A systematic approach needs to concretize the at first rather generic meanings of reliability and economy for the context of this work. The given preconditions, characteristics, aspects and logics provide a basis for a consequential analysis: we are now able to determine reasons for system failures and identify cost-drivers of the OBE software. From there, an appropriate architectural answer to the resulting requirements can be devised. Furthermore, the descriptions of this chapter already hint at specific constraints and conflicts in need of further discussion: complexity versus robustness, proven stability of the software versus variable third-party run-time environments.

The question of how to build stable software systems is still ubiquitous; good solutions abound. However, it is hard to answer on a universal but still practical level if time and money are limited, as is commonly the case. Instead, a domain-specific solution seems pragmatic as well as sufficient to answer the needs of ETC.

Three

Substantiating Domain-specific Reliability and Economy

... in which we describe architectural aims and design decisions; apply the initial requirements defined in chapter 1 to the domain and business processes introduced in chapter 2, elaborating in detail on the meaning of reliability and economy in the context of ETC software. The derived and refined domain-specific requirements determine the software design as well as implementation qualities. Suitable modeling and programming techniques are discussed.

design decisions ...

The elaboration of an OBE software architecture with the aims of reliable and economical operations as stated in chapter 1.2 requires consideration of a number of design decisions. These concern the development process as well as the intended product, ranging from the notation to apply to the definition of structure and behavior to the intended programming technique of the resulting source code. A contextual refinement of the high-level requirements initially defined enables us to reason about these specifics of the HIRTE foundation in the following sections. Each described attribute has to reflect in the requirements and be justified in the light of its contribution to fulfillment.

... and requirements

Note that the following sections label the intended architecture's characteristics "requirements", instead of "design decisions". This is merely to express a certain openness regarding the results of this chapter. We actually interpret the defined requirements as design decisions for the architecture introduced in the next chapters. However, this chapter should also be of use outside the context of this work, providing a set of requirements as guidelines, not hard decisions, for similar solutions.

3.1 Requirements refined

comprehending the aims

On first sight, the aims of economy and reliability seem rather mundane for a software (or perhaps, any) product. After all, this is what every customer expects. What is not trivial are concrete technical measures to achieve these goals, and respective comprehensible proofs of validity. Especially if it is not acceptable for the operator to achieve and prove requirements conformance a-posteriori, i.e. deploying a system and then see if it is running satisfyingly, distributing bugfixes during regular operations to gradually reach stability.

an accentuation

An ETC system adds yet another dimension: as participation is usually not optional for the user, its complex transactions are closely scrutinized. It may be rewarding to challenge the validity of the output of an implementation independently from any actual, observable defects. Any participant is entitled to contest an invoice, e.g. claiming not to have taken the respective route. In this case, a general ISO 9000 certificate of the supplier responsible for the software as proof of correctness may not satisfy a judge.

manifesting the qualities

From the previous chapter we gained an understanding of the operational aspects of the software, and the business processes it has to implement. So the question for manifestations of the qualities economy and reliability in a corresponding ETC architecture arises: how can its specific development, structures and behavior transparently support these aims? To answer this question, we have to discuss the determining attributes of a reliable ETC software, breaking down hazards to stability and cost factors to a technical level. Focus lies, as stated before, on a GNSS/CN "thick client" solution with the OBE computing the identification and charging of toll atoms, as this alternative obviously exhibits the highest complexity of the introduced potential deployments concerning the distributed OBE[1] software. Respective subsets of the following conclusions are valid for "thin client" implementations.

3.1.1 Hazards and Reliability

failure rates

Commonly, governmental authorities issue a set of accepted maximum failure rates (for an example see [MVW05]). From a top-down point of view, it starts with the aggregated system's overall failure rate, to which any element of the process chain may contribute. A decisive characteristic of system acceptance by authorities, market and users, it does not diffentiate between hardware, software, organization or business processes. Further requirements define threshold percentages for invoices that may contain incorrect entries, for on-road tolling transactions that yield incorrect results and general MTBF (*Mean Time Between Failure*) rates for components like the OBU.

accepting imperfection

These statistical rates convey an important message: faulty system be-

[1]Note that the terms *OBE* and *OBU* in this context does not necessarily refer to an integrated device with sensors and communication modules like in Germany. It merely denotes the vehicle-side run-time execution environment.

havior is – up to a defined, low level limited by the corresponding pecuniary damages – unavoidable in a complex system like the given. Consequently, a reliable ETC software design has to emphasize *measures to facilitate stable operations* as well as *mechanisms to respond to problems efficiently.* It is important to acknowledge the fact that these problems may not necessarily be inherent in the software itself. The ETC OBU software however is perfectly situated in the ETC systemscape to detect a range of problems and incidents, i.e. conspicuous events ([Sto96], chap. 4).

The failure rate requirements provide us with an indication of the pursued reliability. We can relate them to the ETC OBU software aspects and take them as an initial lead to risk and cost identification, deriving requirements for the HIRTE architecture on the way.

fault scenarios

Generally, the OBU elements primarily contribute to the overall failure rate of an ETC system with monitored and proven cases of faulty processing of toll atoms. We differentiate five scenarios:

1. The vehicle uses a toll atom while the system completely fails to detect/charge it (commonly denominated as *type I error*).
2. The vehicle uses a toll atom, the system detects it correctly, but charges it incorrectly.
3. The system detects/charges a toll atom while the vehicle is travelling on a toll free segment (*type II error*).
4. The system correctly charges a toll atom, but fails an associated enforcement verification.
5. A genuine fifth fault scenario is introduced by the successful appeal against a correctly charged atom. If a user questions the OBU's integrity and the operator is unable to consistently rebut this claim, another case of failure would be registered.

output of our subsystem

Corresponding to the focus of this work, acknowledgement of a charging event or message is the final step of the ETC business subprocess addressed here. It refers to a system component being notified of a specific vehicle or user liable to pay a specific amount for usage of a toll atom (cmp. chapter 2) and subsequent transaction-concluding reply. This component responsible for further processing and debiting is usually a central billing system (pre- or post-payment mode).

approaching hazards

Thus the question of hazards to the reliability of the ETC software can be narrowed down to an analysis of the aspects (as defined in chapter 2) of our solution in relation to the types of potential error cases defined above, that the operator needs to avoid. The *Fault Tree Analysis* (FTA) approach ([VGRH81] with extensions introduced in [KLM03]; more precisely a *fault graph*) provides us with a structured view on this relation. While similar methods of hazard analysis like FMEA (*Failure Mode and Effects Analysis*, [IEC85]) progressively examine components or functions for possible problems, FTA takes another perspective: starting with a faulty charging processing, equivalent to a primary ETC OBU software failure, we work our way backward, dissecting the processes introduced by the aspects for potential causes of fault events as illustrated in figure 3.1.

FTA

Starting with a general root event, a rectangular node represents a further refinable, or intermediate, fault event. Relations between one node and two or more refining nodes take either AND (&) or OR (≥ 1) logical gate form, implying either all or one or more of the successive events, respectively, as precondition of the intermediate node. A conditioning event can be applied to a logical gate in the form of an oval shape to define specific conditions or restrictions for the associated events. If the condition refers to one event, a hexagon inhibit gate is used. Similarly, two or more intermediate events may be traced back to one single event. As leafs we find triangular transfer-in and transfer-out symbols to indicate further development of a tree in another, correspondingly named diagram. Circle leafs represent basic fault events requiring no further refinement, rectangular leafs with a tip normally expected external events. Diamond shapes indicate undeveloped events. In the given context, originators of these fault events would lie outside of the ETC software's scope – operating system functions, hardware resources, sensors. Consequently, the ETC application can exert very limited or no control at all over these external components.

In our application of the FTA, we are not interested in probabilistic risk analysis, often associated with this method. Aim of the following FTA is a systematic analysis of ETC software hazards to identify and efficiently handle potential problems.

fault scenario 1

Fault *scenario 1* has the vehicle travelling on a toll road or other traffic infrastructure. The corresponding processing of the toll atom however is not completed due to a number of potential reasons. If it is not possible to track a reliable position of the vehicle (cmp. 3.1.1.1 and 3.1.1.7), all successive computations – most notably the matching of the position to a map element representing the toll atom – based on this input become invalid. The same holds true conversely: a valid position match fails if the map's operating data is disrupted (3.1.1.2) or faulty (3.1.1.3), the correct required elements missing or inaccessible. To enforce correctness of transmission, security and to optimize CN load, different transaction protocols are defined between distributed OBUs and central servers. If any inconsistency between the actual charging record message exchange and the protocol implementation was detected (3.1.1.4), the charging process is aborted. Consequently, if the prerequisite for processing of the charging transaction protocol – a stable CN connection to the server – is not met (3.1.1.5), completion of the charging will also fail.

fault scenario 2

The incorrect charging of *scenario 2* refers to a valid matching of the ve-

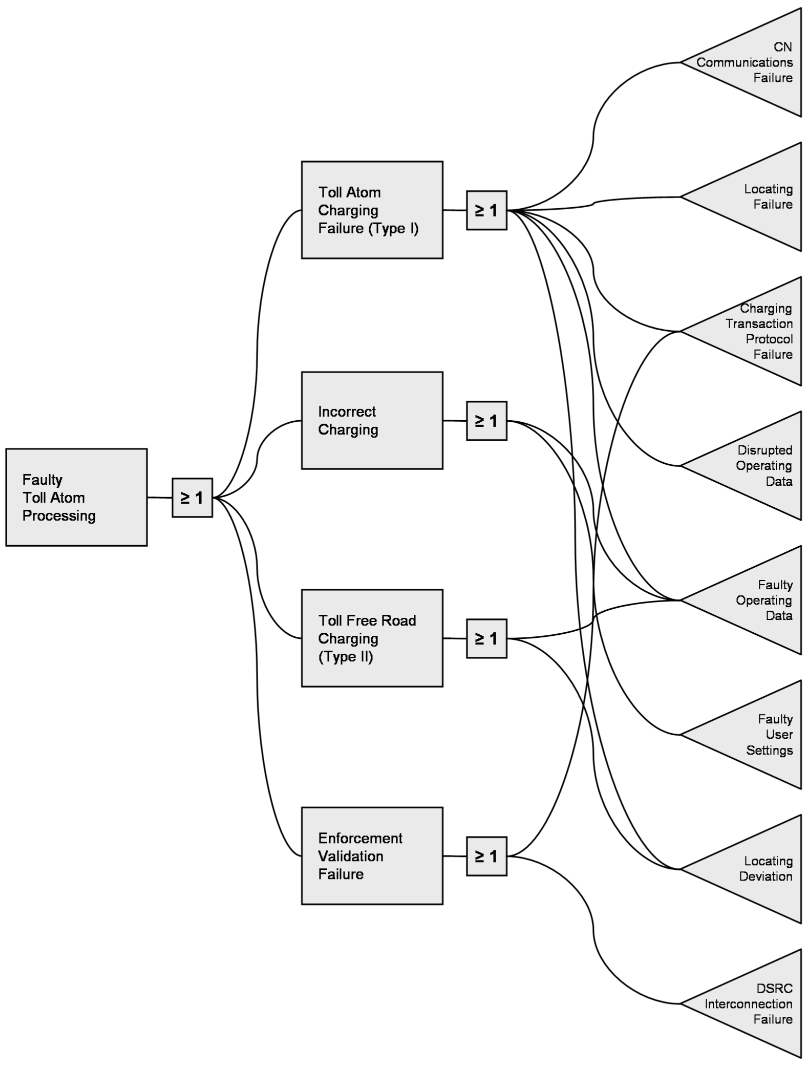

CN Communications Failure
Toll Atom Charging Failure (Type I)
≥ 1
Locating Failure
Charging Transaction Protocol Failure
Incorrect Charging
≥ 1
Disrupted Operating Data
Faulty Toll Atom Processing
≥ 1
Faulty Operating Data
Toll Free Road Charging (Type II)
≥ 1
Faulty User Settings
Enforcement Validation Failure
≥ 1
Locating Deviation
DSRC Interconnection Failure

Figure 3.1: Fault Tree of a Faulty Charging

hicle's position to a toll element, with a subsequent computing of an amount due based on the wrong tariff parameters. These parameters may refer to classifying attributes of the vehicle (e.g. weight, number of axles etc.) as well as the toll atom (e.g. temporal factors, base tariff for usage etc.). The associated values are either entered by the user at the beginning of a trip and stored temporarily, or they are records of the persistent operating database that is updated over-the-air by the servers. Thus, determination of the amount by applying a tariff scheme to the parameters will suffer from faulty operating data (3.1.1.3) as well as faultily declared user settings (3.1.1.6)[2].

fault scenario 3

As the user is initially (financially) damaged by the system charging a toll free road, *scenario 3* has to be regarded as very critical to the acceptance of the implementation. While the operating database is updated and uncorrupted, a sufficiently severe deviation in locating (3.1.1.7) may lead the software to the false assumption that the vehicle is using a matching toll atom and trigger the corresponding charging process. If the operating data is obsolete or was faultily updated, actually invalid or inactive records of toll atoms, temporal validities and tariff schemes could activate (3.1.1.3). The software would then effectively use either outdated or future tolling schemes for charging.

fault scenario 4

Fault *scenario 4* implies a correct charging by the ETC software. However, failing a subsequent enforcement check means that it is unable to prove that in the immediate context. Formally, the charging process could actually still be correctly resolved during regular operations: if the charging records are received by the billing servers, the offender would be cleared later. But as the vehicle might be stopped by the authorities based on the checking failure, the user might unjustly experience a major inconvenience. Thus, the scenario has to be regarded as critical failure. The OBU software element of the enforcement commonly relies on DSRC (infrared or microwave) to carry the transaction. Failure of DSRC interconnection (3.1.1.8) or the protocol (3.1.1.4) implementing the enforcement check results in a failure of the validation of proper charging.

excluding scenario 5

As *scenario 5* is not the result of a technical failure, it will be treated implicitly later in the context of system transparency.

3.1.1.1 Locating Failure

a selective problem

Generally, a failure of accurate positioning affects the charging process only while the vehicle is travelling a toll atom. Beyond that, it becomes a neglectable fault if occurring randomly and seldom. A regular recurrence off toll atom would at least have to be logged as suspicious incident for analysis during the next scheduled maintenance.

application view on positioning

In a DSRC/beacon-based deployment a positioning failure would be equivalent to the failure of the DSRC interconnection (3.1.1.8). A GNSS positioning module has to be regarded as concurrently running process by the OBU software, as positions in a tolling context have to be determined perpetually and independent from other activities of the the application (e.g. communications) to ensure coherent charging. The underlying hardware is actually often an embedded computer of its own (e.g. [Ga06]), providing application-ready data. Alternatively, a corresponding thread of the OBU application uses a GNSS

[2]The case of incidentally reaching the correct sum based on false parameters is likely to be discovered either by trace analysis during testing runs or by a customer complaining about a suspicious bill.

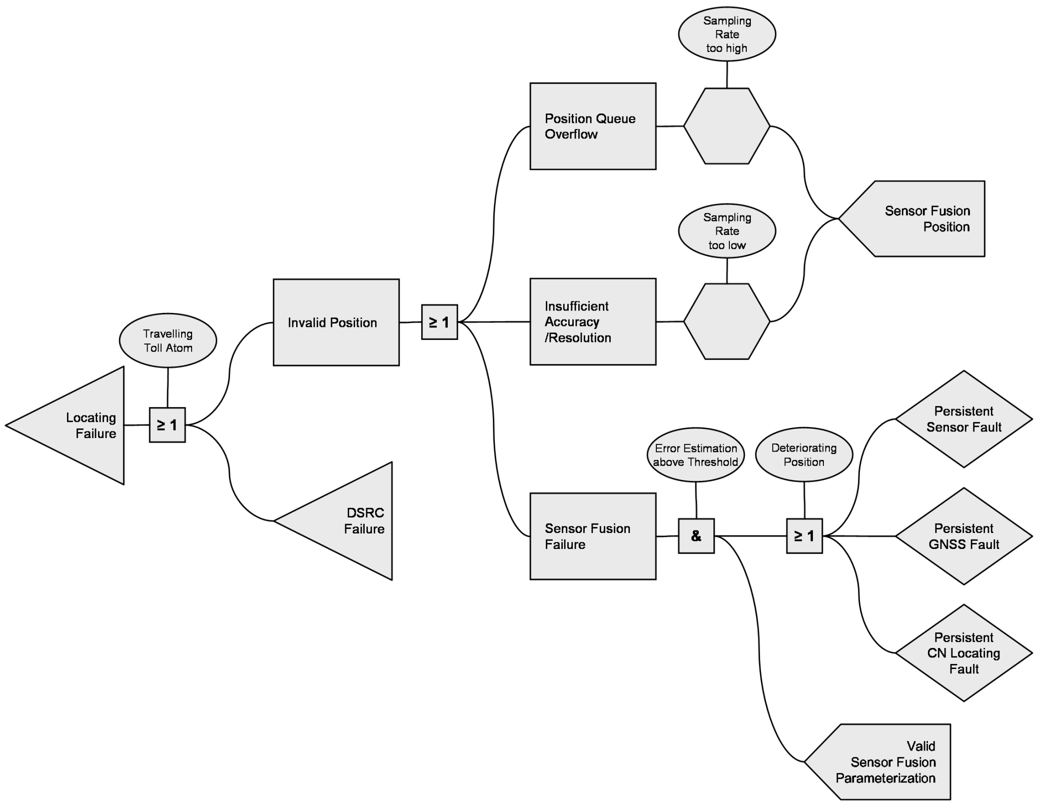


Figure 3.2: Fault Tree of a Locating Failure

device driver to read the satellite signal (and other sensors, see below) for further processing. In both cases, the positioning process continuously estimates a position fix of the vehicle with a specific sampling rate. Consequently working asynchronously, the resulting data is placed in a queue to be dequeued and used by the respective consumer.

violating constraints

While an upper bound to the position sampling rate is given by the device capabilities, a software use case (cmp. chapter 2) dictates the effectively required value. If this value is set too high, the position queue could overflow, resulting in a position loss, or, in other words, an invalid position. From the view of the software, and provided that the high sampling rate is a confirmed requirement, this may either mean that the queue size is insufficient, or that real-time requirements/timing constraints of the positional data computations are being violated. In this case, processing of the positions would lag behind. Reasons for that could be found in an inefficient algorithm implementation or inadequate priority of the corresponding thread of execution. That ruled out, insufficient embedded platform computing power would be the potential source of failure.

position sampling

A locating algorithm requires a certain number of positions to reliably match them to a toll atom. The sufficient sequence of data varies with the type and geometry of the atom, and the speed of the vehicle. If the effective sampling rate is too low in a given situation, gaps between positions become too wide, i.e. the trace resolution degrades. The resulting inaccuracy leads to a diffuse selectivity of toll atoms and other geometry, consequently to an invalid positioning. To avoid type II errors (s.a.) damaging the user, this has to be interpreted as toll free situation.

sensor fusion

To improve the accuracy of positioning, a sensor fusion algorithm (cmp. 2.2.3) accepts a number of input sources to minimize the overall error. Applied properly (i.e. a suitable algorithm, validly parametrized), the sensor fusion will indicate the exceeding of a threshold specifying the maximum tolerable error, marking the current position as invalid. This situation may occur if one or more of the input sensors persistently exhibit faulty behavior. After some time – depending on the actual implementation and relevance of the defective sensor(s) – the algorithm is unable to compensate for the malfunction.

3.1.1.2 Disrupted Operating Data

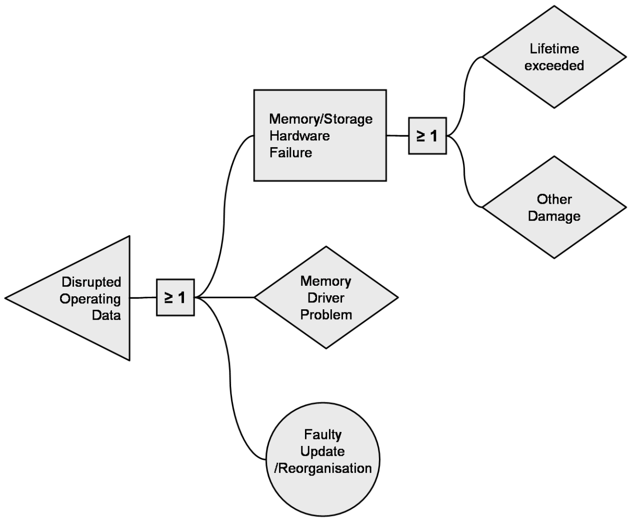


Figure 3.3: Fault Tree of Operating Data Disruption

failed updates

Disruption of operating data generally refers to inaccessible or missing records and associations between them. Potential software sources of faults are update or reorganization activities. An improperly handled update operation may affect a single attribute, structures or an entire dataset release version. Data might be plainly illegaly overwritten or – if a transaction is not coherently executed and interrupted – deleted and not correctly replaced.

reorganization

In an embedded environment usually memory is still an issue. In consequence, consumption by the application has to be optimized, demanding regular reorganization algorithm runs e.g. to avoid fragmentation. Depending on the structure or layer of implementation, either the application itself or an operating system driver is responsible for this procedure. A faulty step or abort, combined with lack of robustness, can lead to corrupted and inconsistent storage locations, disrupting the contained, represented data structures.

memory hardware

On a hardware level, we find that the static storage memory devices required for persistent data may also fail the application. In the case of chip-based memory, the device driver may wear out and damage the hardware. By continually addressing a single specific position or segment for assignments, it can quickly reach the limited number of guaranteed erase-write cycles. Consequently, even with equal distribution the lifetime will be exceeded eventually, with the device becoming unstable and unreliable if not replaced in time. Furthermore, a number of hazards to electronics arise from the automotive deployment, straining and potentially damaging the hardware.

3.1.1.3 Faulty Operating Data

subtlety

In contrast to operating data disruption (s.a.), faulty data in special cases induces more subtle sources of failure, i.e. the corresponding incident might not be represented by a single failed action, but by inconsistencies only detectable in subsequent steps of the charging process. All the more critical becomes the breakdown of potential causes of defect.

center-imported problems

An integrity proven transmission from the central servers may be imported into the OBU data storage while including structurally correct, but faulty information. Validity of the data entered has to be checked by the server processes, as the distributed software generally cannot account for plausibility: it simply has no means. An OBU application will only be able to indicate certain obvious inconsistencies, e.g. an excessively high tariff definition for a short road segment. Thus, centrally cleared distribution of a faulty record or faulty full release of an operating dataset (e.g. an obsolete version accidentally tagged as current) will eventually lead to a faulty charging.

premature OD transitions

A complete dataset will become obsolete, rendering the OBU unable to process any charging, if a release transition occurs before the superseding release is received and imported by the application. The transition may in this case refer either to a chronological (physical changes to the infrastructure, changes in tariffs etc.) or regional (the vehicle travelled to a new map segment, city, country etc.) update.

distribution management

An underlying failure of update distribution can have a number of reasons. Problems with cellular communications (3.1.1.5) can prohibit the nec-

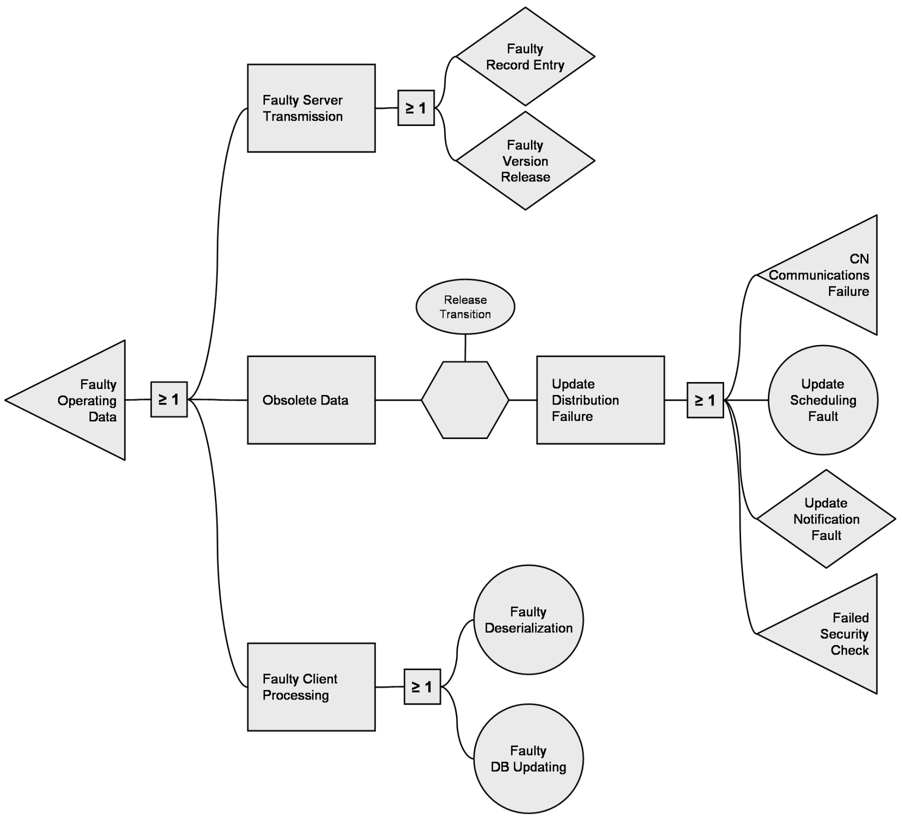


Figure 3.4: Fault Tree of Faulty Operating Data

essary connection to a central server. If updates are scheduled and initiated by the OBU software, faults in the corresponding timetables, errors in random distribution algorithms (or their parameters), inactive update threads or lost events/signals make the application miss the allocated time slot. The same holds true if the update transaction is centrally initiated, with the server failing to notify the distributed OBUs of an upcoming release. Furthermore, the received messages of a successful update transmission have to be security checked. If this action fails (3.1.1.9), the application is not allowed to import and activate the new operating data.

message handling

The client OBU itself introduces a range of problems into the charging process, if the handling of accepted messages is faulty. Deserialization of records with attribute order and values has to be consistent with the server-side encoding. A rather basic case of failure is given with plainly undecodable or incongruous data, e.g. noticeable type violations – it may eventually lead to an obsolete dataset (s.a.), but identification of the source should be apparent. But furthermore, the deserialization could permute the order of values of the same type, interchanging assignments to record attributes, respectively. It could misinterpret the bit order of values, conforming to the proper type while producing wrong values. If the resulting span between expected and falsely decoded values is only slight, a plausibility check might not detect the inconsistency, potentially resulting in very subtle anomalies of the charging process.

importing data

Depending on the strictness of the database management implementation, similar problems may arise when the decoded data is imported into the stored datasets, as records have to be serialized again for this purpose. Even if the updating process handles de-/serialization and assignments correctly, it could still associate the set with a wrong release version, effectively either deleting or abandoning the corresponding release. This would eventually result, again, in an obsolete dataset for the specific period. Equivalent risks have to be considered regarding database reorganization procedures.

3.1.1.4 Charging Transaction Protocol Failure

invariant security

When the client OBU software communicates with the central servers during the charging process, the transaction adheres to a specified protocol, defining sequence, timing constraints and types of messages. As described in chapter 2, security is an integral element of the software and its activities: each transaction protocol has to conform to an underlying security protocol. Consequently, a transaction fails, if in any step a security constraint should be violated (3.1.1.9).

protocol inconsistencies

If at any time during an (over-the-air) exchange expected and received messages contradict each other, the software is confronted with an inconsistent protocol sequence and correspondingly an implementation inconsistency between OBU and server application. This inconsistency may be founded in a buggy or specification-nonconforming operation of the client or server program. Lack of coordination and alignment between distributed and central software releases can also imply a potential collision of protocol implementation versions. Reception of an unknown message type is a particular case of protocol inconsistency, basically with equal underlying faults.

lost messages

A transaction has to be regarded incomplete, the corresponding charging failed, if the server does not explicitly confirm the sent charging records. The central system may either reply with a rejection of the transmitted and actually received records, or not reply at all. In this case, after a specified period and depending on the kind of cellular network service, a communication timeout will occur. If the timeout condition recurs over a defined number of subsequent retries, the transaction is failed.

lost records

The access-, availability and integrity of OBU-processed charging records is a prerequisite for the OBU software to generate the charging record messages to convey to a central instance for further processing and billing. Charging records usually have to be buffered, as the application collects them until a

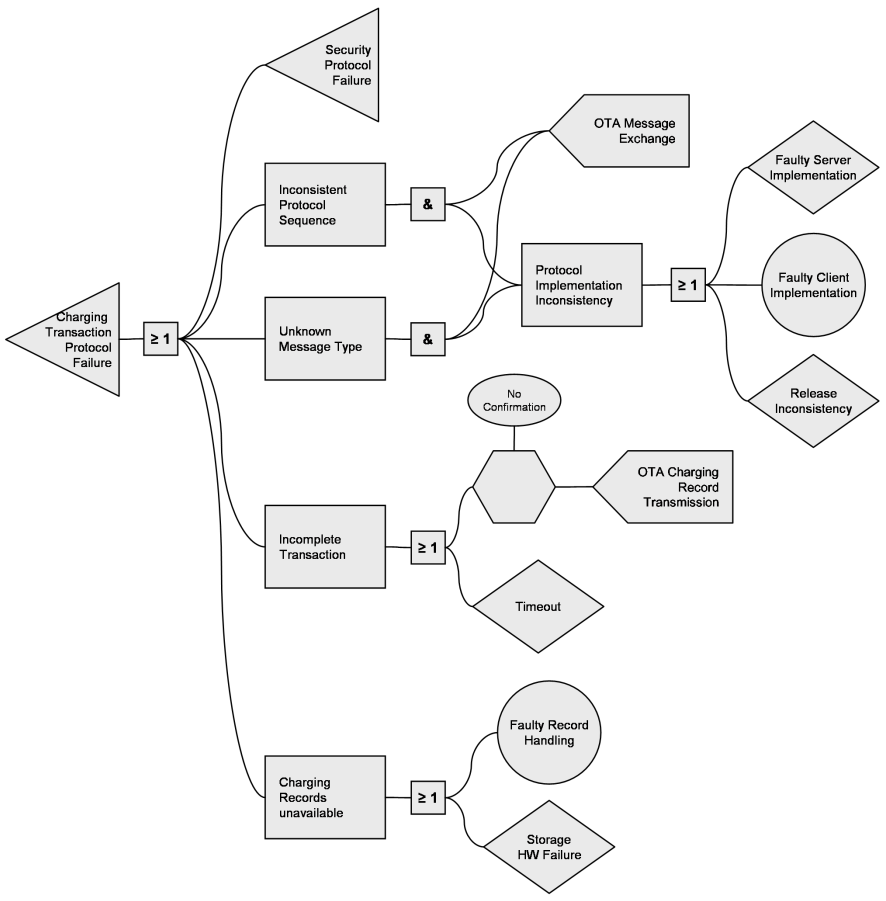

Security Protocol Failure
Inconsistent Protocol Sequence
&
OTA Message Exchange
Faulty Server Implementation
Protocol Implementation Inconsistency
≥ 1
Faulty Client Implementation
Charging Transaction Protocol Failure
≥ 1
Unknown Message Type
&
Release Inconsistency
No Confirmation
OTA Charging Record Transmission
Incomplete Transaction
≥ 1
Timeout
Faulty Record Handling
Charging Records unavailable
≥ 1
Storage HW Failure

Figure 3.5: Fault Tree of a Charging Transaction Protocol Failure

transmission to the server is scheduled, e.g. to minimize CN usage. Similarly to 3.1.1.2 and 3.1.1.3, handling and storing of these records may fail.

3.1.1.5 CN Communication Failure

For interactions between distributed and central units, the system relies on the availability of a cellular network and its services. General quality of service considerations can be decomposed with regard to the OBU application.

inaccessible CN devices

The software may find the respective device (e.g. a GSM modem) and the associated service (e.g. SMS, BS 26, GPRS, UMTS; cmp. [EVB01], chap. 4) unresponsive in the first place. A rather trivial case of broken hardware is complemented here by potentially inconsistent modules: it is possible that either the software version does not correspond to the installed device, or that the application software does not work with the provided device driver.

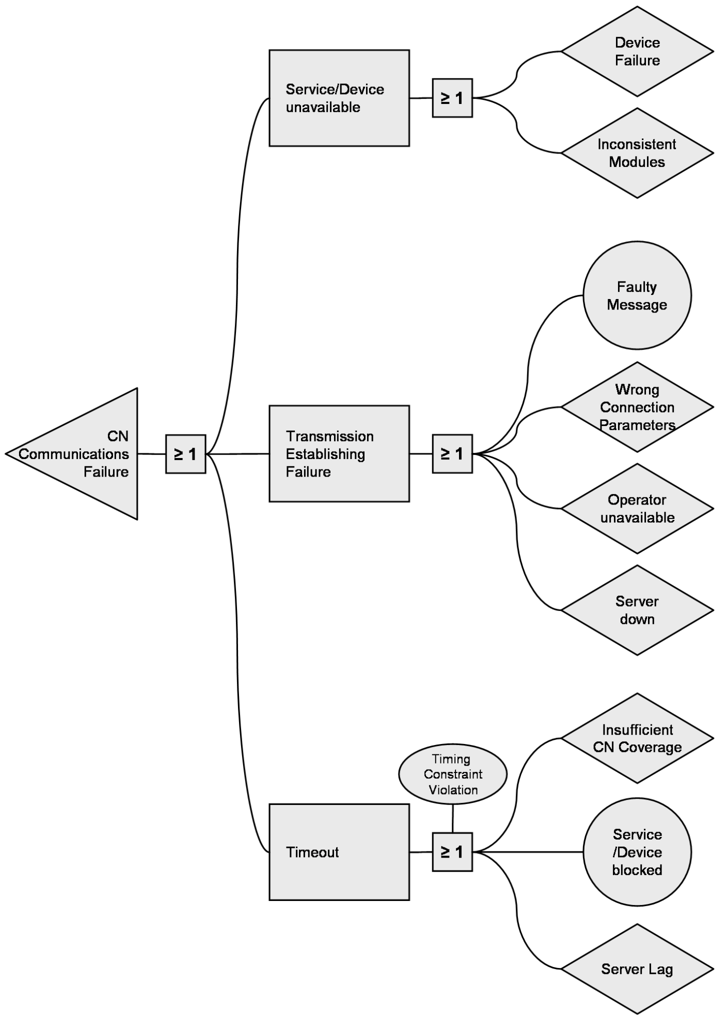


Figure 3.6: Fault Tree of a CN Communication Failure

no CN interaction

Given a responsive CN module, establishment of the transmission will fail, if the passed message itself was constructed incorrectly by the application concerning length, structure, attribute/field formats or termination. As a prerequisite to contact a server, the OBU requires a set of communication parameters, e.g. MSISDN ([EVB01]), IP addresses and additional configuration specifics depending on the type of connection. Erroneous or obsolete definitions of any one of these parameters will very likely result in a transmission initialization failure. In contrast, the case of a correctly defined but unavailable CN operator may be related to lack of network coverage, either due to general infrastructure gaps or the vehicle travelling outside of the area bounded by a quality of service agreement (e.g. beyond the borders of a country). Furthermore, the distributed unit may find the server element down for irregular reasons.

enduring timeouts

After a defined number of retries following a connection timeout event, the CN communication is failed. This CN timeout refers to a physically established service connection and has to be differentiated from the logical timeout of 3.1.1.4 during message interchange. Thus, the link between OBU and server can be temporarily terminated by insufficient CN coverage and signal distortions in a given area, handover problems during travel, or a stressed, lagging server system. On the side of the OBU, if the resource CN device is shared between threads of the OBU software, enduring or permanent allocation by one thread and subsequent but denied request by another may lead to the timeout of a CN connect action attempted by latter thread.

3.1.1.6 Faulty User Settings

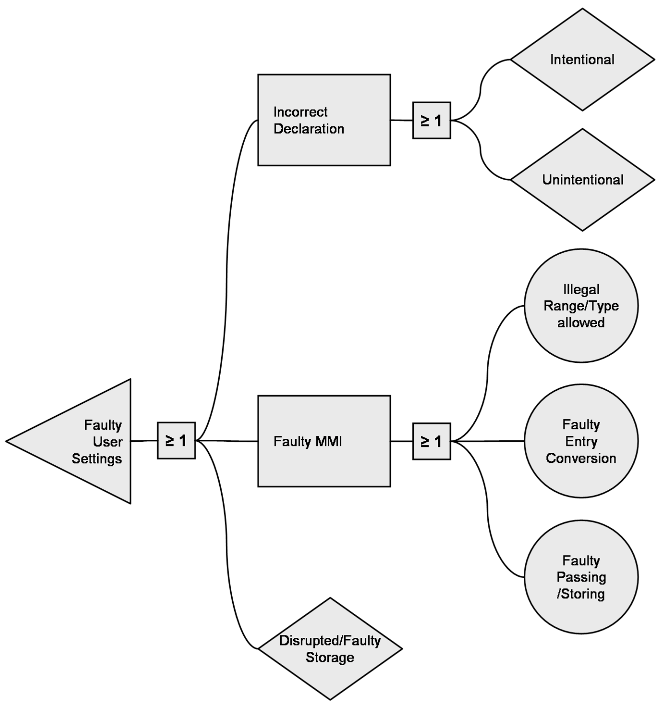


Figure 3.7: Fault Tree of Faulty User Settings

sensitive questions

The topic of user settings in relation to an ETC system touches the very sensitive aspect of the user's obligation to co-operate (cmp. beginning of 3.1). In the case of a non-technically induced incorrect declaration of charging parameters, the question whether it was intentional or unintentional arises, and if technical measures (e.g. plausibility checks) could have prevented or proven it.

MMI complexity

Especially with a multi-language, multi-character set human machine interface (HMI; or man machine interface MMI), conversion of country-specific entries to internal value assignments becomes an issue. On a more general level, ranges and types of user entries have to be checked after conversion from string or list items. The same may be required during the process of storing the input data.

To compute and generate charging records the application has to access, among other values, the user setting parameters – they effectively become part of the operating database of the OBU software. Consequently, similar potential faults have to be regarded (cmp. 3.1.1.2).

3.1.1.7 Locating Deviation

off-atom effects

In contrast to a locating failure recognized as such (cmp. 3.1.1.1), a locating deviation presumes position data indicated as valid. But unlike the detected failure, it may have an effect on the charging process even if the vehicle is not travelling a toll atom (cmp. type II errors above).

map matching

In a GNSS-based ETC system, positions have to be mapped to geometrically described toll atom structures (cmp. 2.2.1). While the corresponding algorithm can be loosely or tightly coupled with a sensor fusion, map matching introduces genuine potential faults into the process. An incorrect mapping may be the result of errors in the implementation as well as improper parameter values, like too high/low tolerance or selectivity thresholds. For a given scenario, a map matching approach itself can prove conceptually false: an efficient heuristic applicable on motorways may be unfit for an urban environment.

combined problems

The combination of a faultily parametrized sensor fusion filter and a deteriorating quality of position may yield an irregular failure. In this situation, the fusion algorithm is incapable to notice and identify a progressive failure of one or more positioning or dead reckoning sensors. Due to incongruous parameters, it misreads the sensor values, thus introducing a defect into its computations and passing on the flawed result as valid[3].

DSRC scatter

In a DSRC-based locating scenario, a deviation in locating may be triggered by the sender's signal scattering beyond the intended limits of communications, e.g. to a road running close to the toll atom structure.

3.1.1.8 DSRC Interconnection Failure

Interacting with roadside infrastructure via DSRC exhibits some hazards similar to CN communications. Others are genuine, related to the fact that the

[3]The case of a deteriorating position induced by other, inconspicuous reasons than sensor failure and the case of a faulty parametrization distorting valid sensor readouts are implied. They both lead to the same conclusion – an unflagged but invalid position.

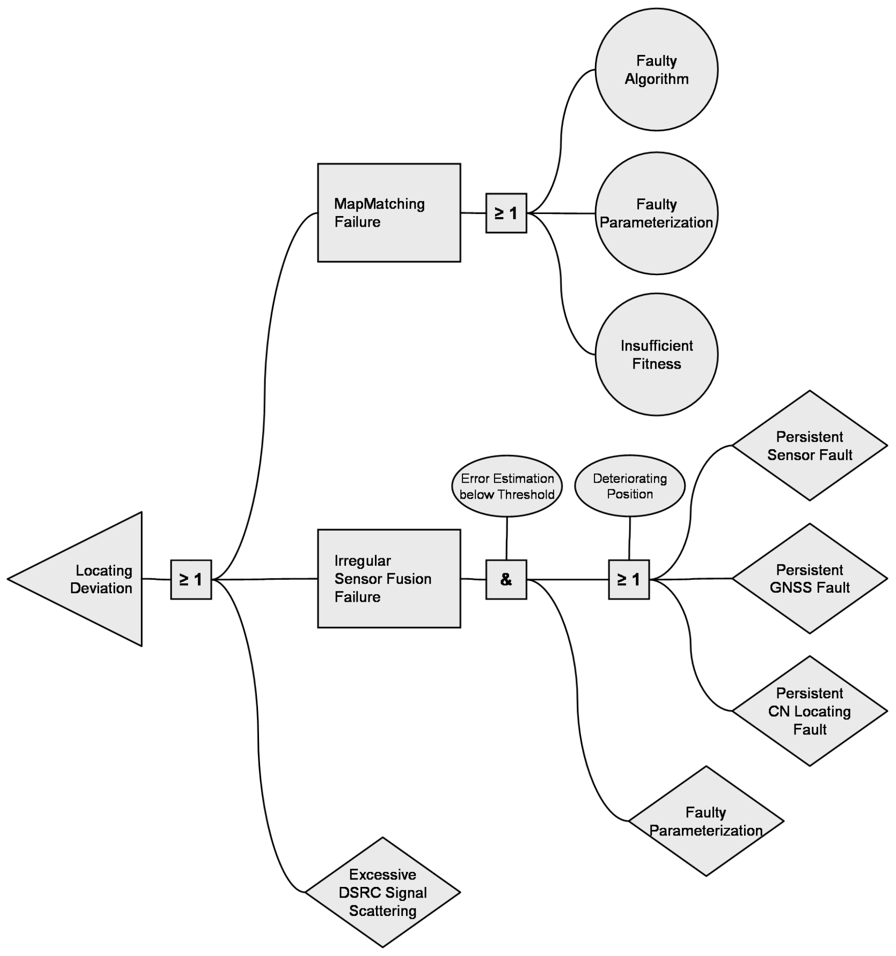


Figure 3.8: Fault Tree of a Locating Deviation

transaction between moving vehicle and fixed installation is constricted by a narrow window of opportunity.

Consequently, here also (cmp. 3.1.1.5), device failure or hardware/software inconsistencies may lead to the OBU software being unable to access the DSRC module. One step further, an incorrectly constructed message, invalid parameters of the connection definition or a shut down DSRC beacon will prohibit the establishment of the transmission.

DSRC-specific constraints

While a timeout of a message may be the result of a service blocked by another thread of the application or a signal distortion, in contrast to a CN scenario, a given time constraint exists for the completion of the transaction itself. This constraint may be violated in relation to the speed of the vehicle passing the DSRC installation. With respect to feasibility and acceptability

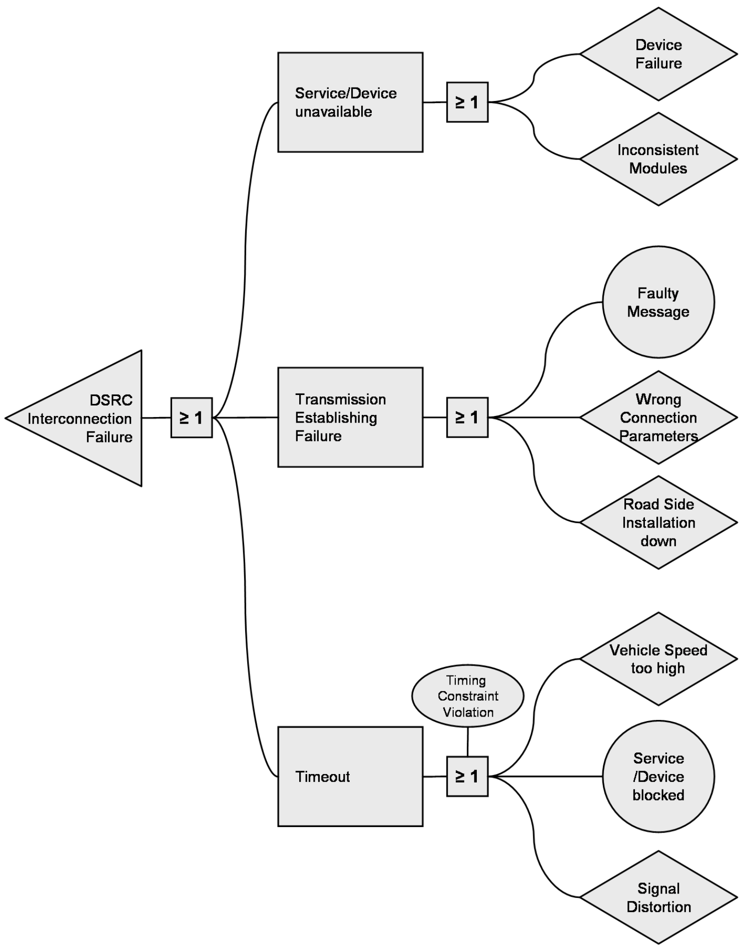

DSRC Interconnection Failure
≥ 1
Service/Device unavailable
≥ 1
Device Failure
Inconsistent Modules
Transmission Establishing Failure
≥ 1
Faulty Message
Wrong Connection Parameters
Road Side Installation down
Timeout
Timing Constraint Violation
≥ 1
Vehicle Speed too high
Service /Device blocked
Signal Distortion

Figure 3.9: Fault Tree of a DSRC Interconnection Failure

of the system, this is a critical scenario: the domestic speed limit determines a minimum time interval available for processing and concluding the transaction (the domestic limit would be replaced by a technical limit on German motorways). If a charging failure occurs in the context of a DSRC timeout, the OBU software has to support anwering the question whether the run-time execution was too slow or the driver was speeding.

3.1.1.9 Security Protocol Failure

Security being an integral part of all ETC processes, the OBU software implements the standard functions to authenticate, encrypt, decrypt and check integrity. Consequently, any of these actions may fail during operations. As the mechanisms (today) can be safely assumed to be based on public key cryp-

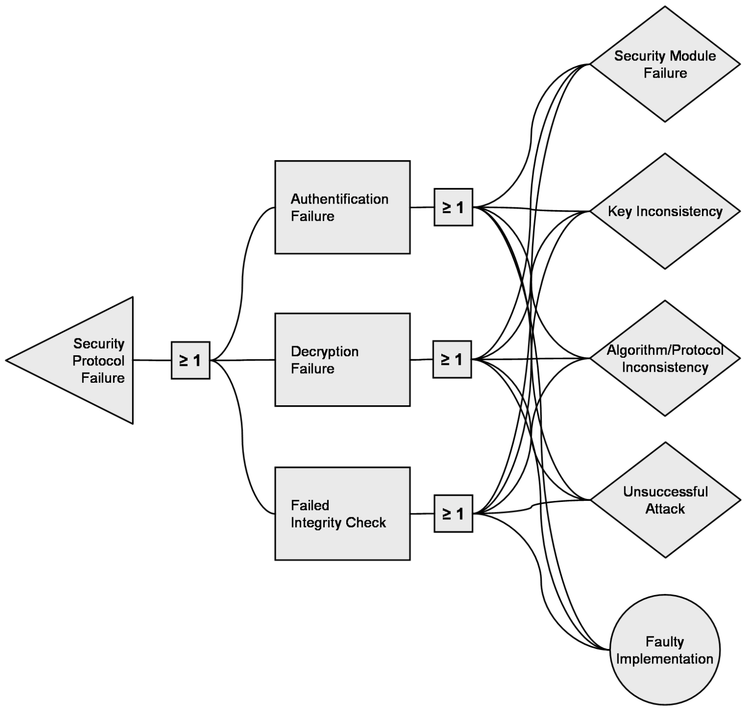


Figure 3.10: Fault Tree of a Security Protocol Failure

tography, their crucial parts encapsuled in one security module (e.g. a smart card), we can expect equivalent or at least closely related reasons for failure in each case.

authentification

An authentification failure can occur on the OBU as well as server-side: whenever a connection has to be established, or a message is received, sender and receiver have to identify and authenticate themselves (cmp. [Sc96], chpt. 21). If a charging record set is received by the center systems, it has to be proven that an authentic OBU – registered with the operator – is the sender. Conversely, the OBU software has to ensure that it is sending its revenue-effective records to a legitimate server.

integrity of process steps

The same holds true for integrity checks. On reception of a message, both OBU and server require, as a precondition for further processing, that no attacker tampered with its content. If the checking of a transaction step fails, the corresponding activity has to be canceled.

decryption

A failure of data decryption is generally more likely to affect the server system. This is based on the fact that the necessity for privacy primarily concerns information sent from OBU to center: charging records or positions of the vehicle have to be regarded as related to an individual person and therefore have to be protected.

basic failure

Any security operation will fail if the underlying module malfunctions or fails. The respective device as well as connectors (e.g. a bus of any kind) between the OBU software platform and the security hardware may be subject to defects in this case.

key inconsistency

Faulty scheduling of updates can lead to an exceeded lifetime of a key, rendering the OBU software incapable of authenticating itself anymore due to inconsistent key validities between OBU and center. Additionally, operational respectively administrational problems (e.g. in the certification authority) may interfere with the processing of key distribution, resulting in inconsistent key pairs.

implementation inconsistency

Algorithm and protocol implementation inconsistencies can become an inter-component issue (e.g. between OBU and server side), and especially when it comes to international interoperability between domestic system installations. While common cryptographic procedures are largely standardized, different interpretations of specifications or proprietary modifications might still be discovered, e.g. concerning the computation of (intermediary) results or sequence of actions.

attacks

As consequence of a successfully repelled attack, the defending security operation will fail, recognizing the illicit input. The associated transaction will be subsequently terminated.

fault identification

Potentially, it may be hard to differentiate between the effects of the faults described above and failures caused by a plainly deficient software implementation, i.e. source code bugs. Here, the degree of subtlety – and thus, the costs to eliminate the problem – depends on whether an operation explicitly fails or completes, producing a flawed result.

3.1.1.10 Hazards Generalized

The fault tree analysis of the previous paragraphs takes the processes introduced in chapter 2 into consideration, highlighting potential flaws. While an actual deployment may feature a range of additional vulnerabilities, the described set of hazards is sufficient to allow us reasoning about a generalization of conceivable problems in the context of the OBU software. Here, realization of the aspects has a number of dependencies, which might be subject to failure.

system area/level of hazard manifestation

Update notification faults, release inconsistencies and the issue of wrong connection parameters can be examples for problems in *organizational* respectively *operational* processes. As an ETC system calls for a significant administrational overhead, mistakes on that level eventually affect the technical software process level. Additionally, the ETC deployment relies on defined, mandatory *quality of service* agreements of its *infrastructure* elements. Unmet requirements may result in invalid GNSS positions and unstable CN connections. Mismatched positions, invalid information and incorrect amounts may hint at faulty data: all processes rely on the *validity, integrity and timeliness of operating data.* Finally, any aspect can be affected by faults of the *implemen-*

tation itself. From a certain level of maturity of the software on, this should be the least likely origin of a failure. However, a software fault – in coding, memory management, concurrency, logics – may represent the root of any of the problems described above.

system element of hazard manifestation

Hazards may manifest themselves in any element of the deployment and their relations: the ETC software itself, the actors associated with it (its underlying hardware with memory, sensors and communications interfaces; roadside and server-side infrastructure, third-party services like a cellular network), or the interfaces between them.

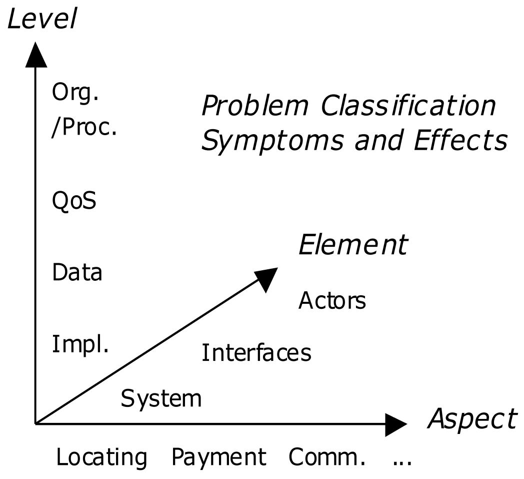


Figure 3.11: Generalization of Hazard Correlation and Classification

The overall, generalized correlations are illustrated in figure 3.11. We can apply these criteria to systematically classify the mission-critical hazards gained from the previous analysis as shown by figure 3.12. The assignment of a fault to a system element is indicated by angular brackets.

lessons learned so far

For a hazard-based deduction of actual requirements for the architecture, we come to a conclusion. The system is highly interactive: actors related to the OBU software are diverse in interface, behavior and degree of integration (e.g. tightly as sensor, or peripheral as billing system). In consequence, the multitude of potential fault originators effectively denies a closed elimination of hazards – even those considered mission-critical, leading to system failure – by finding all faults of the ETC software itself[4]. Many problems may plainly stem from outside of the scope of the application (cmp. figure 3.13), making a risk assessment (i.e. determining hazard likelihoods and safety integrity

[4]At least hypothetically this is achievable, if all else fails by "brute force": allocating unlimited resources on testing this element.

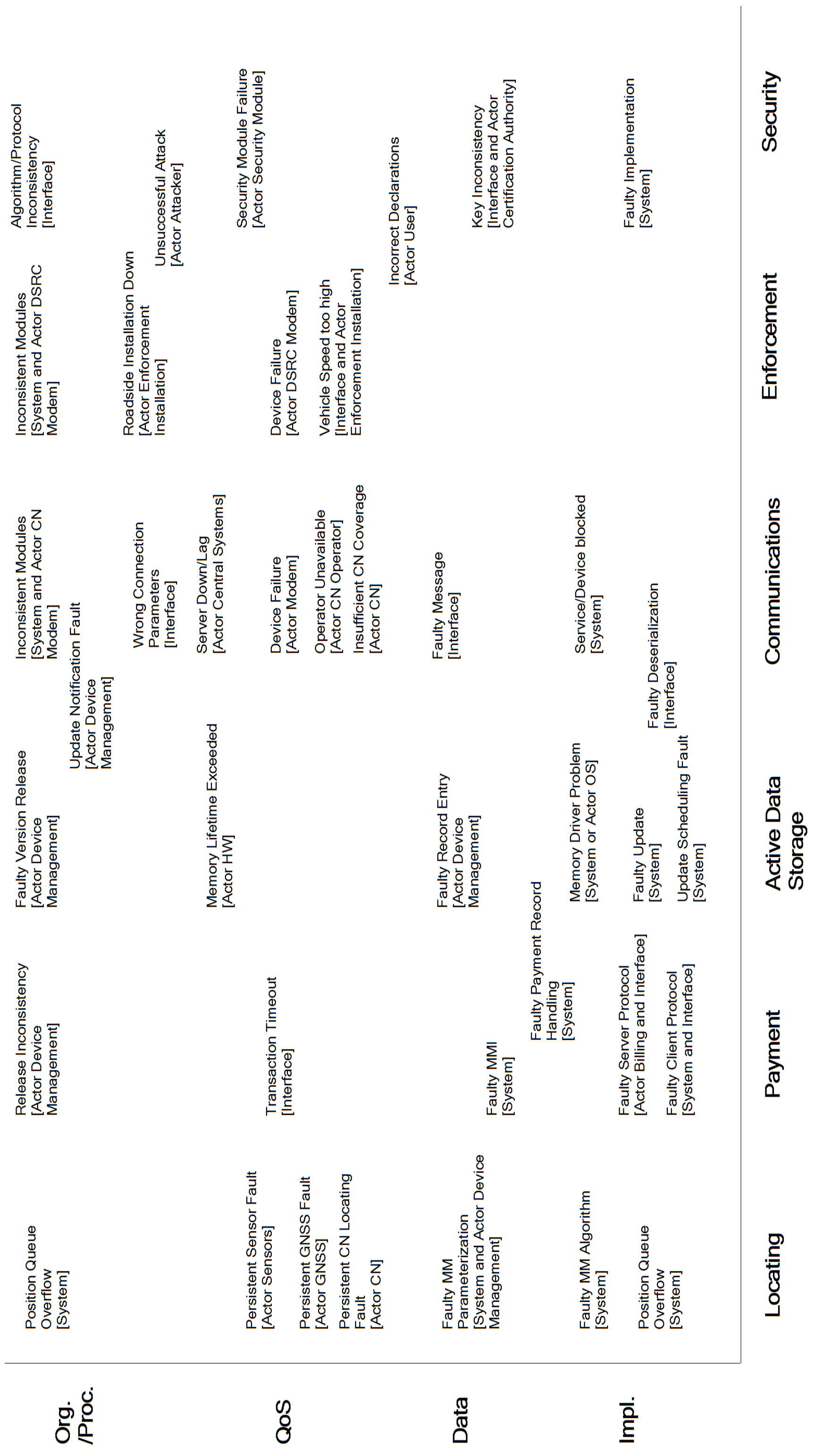

Org. /Proc.
QoS
Data
Impl.
Locating
Payment
Active Data Storage
Communications
Enforcement
Security
Position Queue Overflow [System]
Persistent Sensor Fault [Actor Sensors]
Persistent GNSS Fault [Actor GNSS]
Persistent CN Locating Fault [Actor CN]
Faulty MM Parameterization [System and Actor Device Management]
Faulty MM Algorithm [System]
Position Queue Overflow [System]
Release Inconsistency [Actor Device Management]
Transaction Timeout [Interface]
Faulty MMI [System]
Faulty Payment Record Handling [System]
Faulty Server Protocol [Actor Billing and Interface]
Faulty Client Protocol [System and Interface]
Faulty Version Release [Actor Device Management]
Update Notification Fault [Actor Device Management]
Memory Lifetime Exceeded [Actor HW]
Faulty Record Entry [Actor Device Management]
Memory Driver Problem [System or Actor OS]
Faulty Update [System]
Update Scheduling Fault [System]
Faulty Deserialization [Interface]
Inconsistent Modules [System and Actor CN Modem]
Wrong Connection Parameters [Interface]
Server Down/Lag [Actor Central Systems]
Device Failure [Actor Modem]
Operator Unavailable [Actor CN Operator]
Insufficient CN Coverage [Actor CN]
Faulty Message [Interface]
Service/Device blocked [System]
Inconsistent Modules [System and Actor DSRC Modem]
Roadside Installation Down [Actor Enforcement Installation]
Device Failure [Actor DSRC Modem]
Vehicle Speed too high [Interface and Actor Enforcement Installation]
Incorrect Declarations [Actor User]
Algorithm/Protocol Inconsistency [Interface]
Unsuccessful Attack [Actor Attacker]
Security Module Failure [Actor Security Module]
Key Inconsistency [Interface and Actor Certification Authority]
Faulty Implementation [System]

Figure 3.12: Classification of Potentially Mission-critical Problems

requirements, cmp. [IEC05], 3.2 and 3.3) of our identified hazards obsolete. It is important to note that accepting this statement does not imply to relax the conventional measures of quality assurance, neither does it argue toward contentedness with flawed products. Instead, we now have gained a basis for a systematic, robust and constructive handling of problems, including critical ones.

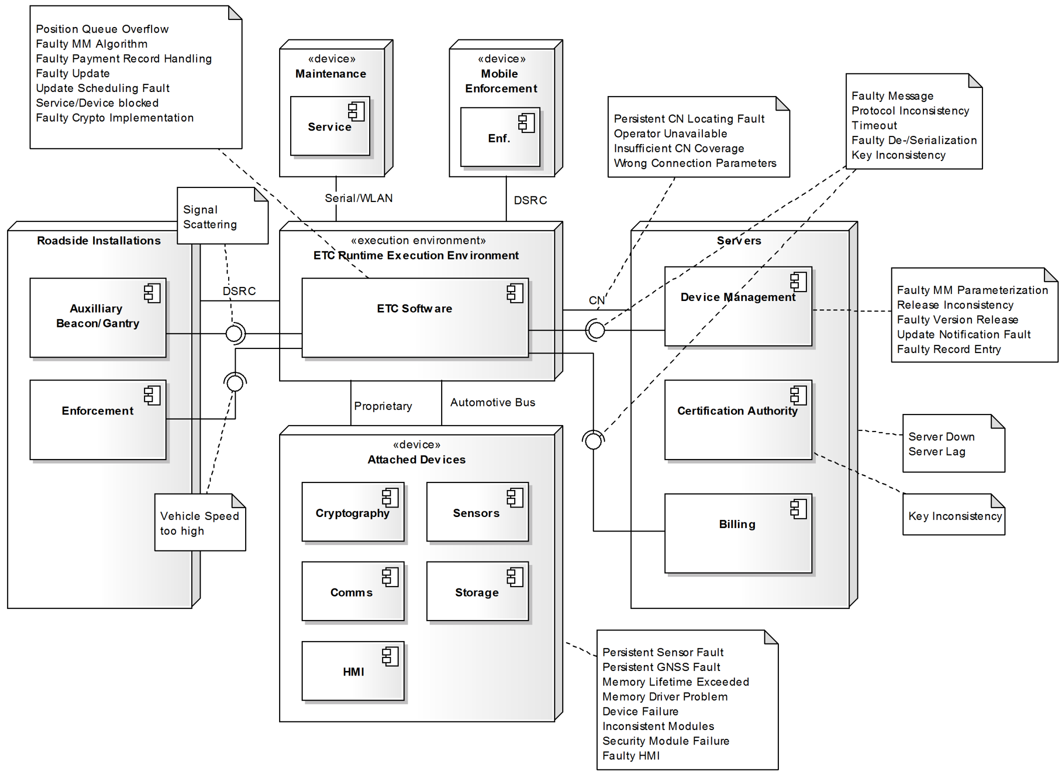


Figure 3.13: Identified Hazards in Relation to the Deployment

3.1.2 Hazard-derived Requirements

an altered view on fault tolerance

To meet our initial claim not only to minimize faults, but to identify, locate and handle the remaining ones efficiently, we slightly alter the concept of fault tolerance that is primarily concerned with the redundancy-based (cmp. [Sto96], chap. 6) or self-reconfiguring (e.g. [RW05]) avoidance of system failure. While it is still desireable to compensate for failures whenever feasible (see the sensor fusion above for an example), the options are rather limited in the given situation, as redundancy is generally expensive and malfunctions of an ETC system do cost money, but do not endanger human lifes.

demands of redundancy

We find many examples that present measures to ensure reliability as cost drivers. Redundancy as a basic principle requires more resources than actually necessary to ensure functionality. Hardware elements (computers, sensors, etc.) have to be installed multiple times. Redundant software modules imply multiplication of the associated run-time hardware resources (processor, memory, interfaces) to be effective. Additionally, failure management becomes not just an issue of shutting down systematically, but seamlessly switching to another part of the system during operations. This results in higher software complexity, e.g. due to the required synchronisation mechanisms, and necessarily higher development and validation costs.

fault awareness

Alternatively, we set the focus on *fault awareness* and maximum transparency, facilitating a rapid selective analysis of problems. A related approach is described with the *availability tactics* of [BCK03] (chapter 5.2). This will also benefit the testing and integration phases by pinpointing and isolating trouble spots – in consequence, more faults may be discovered and resolved in a given time, stabilizing the system prior to deployment.

problem manifestations

To systematically handle actual error manifestations based on the generalizations of the previous section, we have to classify their effect and symptoms relating to the OBU software. In this context, the severity of a problem determines much of what will be remaining to analyze after the occurrence.

Total Crash – a plain OBU software crash completely halts all processes of the application. It is by any means unrecoverable and cuts off all interaction with the OBU element. In this case, effect equals symptoms. If any artifact remains for analysis, depends on whether the crash affected or was caused by the hardware, e.g. physically damaged memory would preclude the retrieval of an error log.

Relevant information for resolution: the system state(s) leading up to the crash, including variable values, contents of message buffers and last process activities, potentially hinting at the problem's origin.

Example: undiscovered memory constraint violations leading to arbitrary address space overwriting.

Critical Error – OBU application functionality and integrity can neither be maintained nor recovered without outside intervention. An OBU system element – hardware module, process/thread, data record including messages – produced/contained invalid results or was unavailable/did not reply at all, if applicable regarding a defined number of retries.

Symptoms: business process becomes incoherent, the chain of events is broken or inconsistent with nominal condition, fatal error messages.

Example: a defective GSM modem suppressing sending and acknowledgement of tolling records, deadlocked threads.

Recoverable Error – the software raises a tolling failure according to 3.1.1. An OBU system element or process step fails temporarily, but is still responsive and eventually continues to operate. Based on the current system state, measures to confine the loss are defined and applicable.

Symptoms: chain of business events is temporarily broken, regular error messages of the application, one-time failed plausibility checks.

Example: temporal GPS failure in urban environment.

Suspicious Incident – problems that do not immediately collide with the ETC business process logics, but potentially converge toward error conditions. A missing/corrupted result that can be reproduced, retransmitted or interpolated.

Symptoms: regular warning level messages of the application, sequences of values approaching plausibility thresholds, isolated deviations from nominal conditions.

Examples: increasing number of GSM modem timeouts and retries, continually dropping free memory, uncritical but recurring positioning errors.

centralized control requirement

The descriptions of the previous sections illustrate that the origins of faults potentially occurring in the context of the OBU software are manifold. To initiate a management hub to handle the multitude of activities, we deduce

Requirement I (centralized control): a dedicated active element shall monitor and orchestrate the processes of the software effectively.

Note that we can accept a potential *single point of failure* (SPOF) here, as 2.3.8.4 and the previous section already declared a tight integration of processes and little room for redundancy. In consequence, either *all* crucial components are available, or the entire application fails, i.e. the larger number of modules represent SPOFs, anyway. Running the application without a central control element especially would imply lack of coherent orchestration and therefore intransparent transactions.

system state automata requirement

Furthermore, it should have become clear that detection of the symptoms goes beyond identifying single event occurrences. Rather, it requires consideration of full system state configurations and progression of states, values and events over time. Thus

Requirement II (system state automata): the software processes shall adhere to a concept of state machines (cmp. e.g. [Gi62]), making explicit and resolving their behavior into discrete states.

This way we approach a unification of the behavioral descriptions that is formalizable and makes the processes implemented by our architecture comparable.

To be able to utilize the information of the state machines during run-time, we define

activity transparency requirement

Requirement III (transparency of activities over time): for any point t in the real-time span observed since system initialization, there has to exist a complete and timely system state and data configuration.

Implied is a tracing instance, instantaneously updated after the transition of a state machine into a new state. Beyond generally providing a detailed view on the software behavior, the resulting trace log can identify potentially hazardous state configurations or state sequences, e.g. recurring (but still recoverable) fault states of a degrading component.

By conforming to these requirements, the OBU software can actively and systematically provide the means to support the controllability of hazards: an exploitable transparency of processes leads to the rapid identification of problems in all phases of the software lifecycle and consequently to solid reliability.

3.1.3 Costs and Economy

The requirement of efficiently handling potential hazards to operations, as introduced in the previous chapter, addressed an important parameter of the overall cost structure of the ETC system. It directly influences the efforts for maintenance and upkeep. With the departure from redundancy, another substantial cost driver was excluded from our considerations.

The question remains for further technical means to optimize the operational cost structures, i.e. what amount has to be spent on which component of the system. From a wide range of possibilities we identified three cost-driven approaches fitting the context of this work. They all aim at avoiding the necessity of dedicated hardware infrastructure, in the process optimizing usage of resources. But – from the perspective of the ETC application – they take different paths.

3.1.3.1 Approach I – Opening the ETC Infrastructure

Deployment of an ETC system includes the installation of a broad basis of telematics units in vehicles and the establishment of communication channels between distributed and center components. This is complemented by the ensurance of availability and service quality of both components and interfaces.

Earlier, we explained that the high effort is justified in the light of the amount of fees levied by the ETC operation, and the consequent risks involved

universally usable infrastructure

for the operator. However, there are other applications that would benefit from the ETC infrastructure and its quality, without themselves being able to cover the associated costs. We find a number of these telematics services that can be partially mapped to a subset of the ETC aspects defined in chapter 2.3. Two examples from the domain introduced in section 2.1.2 will illustrate this.

tracking and tracing example

Tracking and tracing use cases describe a vehicle sending a message to a central server on a regular basis, with the center or OBU side initiating communications, respectively. The information encoded in the message generally has to be considered confidential. It might include the vehicle's and thus the driver's position, the payload in case of a truck and various operating figures concerning the state of the vehicle; motor, fuel, tires. Consequently, each message has to be encrypted. Depending on the extent of information to convey, additional interfaces or sensors might be required. ETC commonly does not include e.g. measuring tire pressure, or reading out fuel consumption or engine hours over CAN via FMS (Fleet Management Standard, [FMS05]). In other cases, the service process may completely rely on the ETC resources, e.g. for cryptography, position and vehicle ID, with no further need for specific extensions. In any case, the implementation of the service requires less dedicated investments, if integrated into the ETC environment (cmp. fig. 3.14).

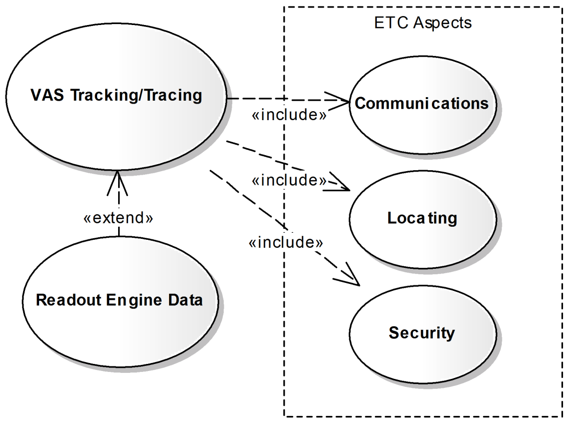


Figure 3.14: Value Added Services Use Case Structure Example

logistics support example

Autonomously controlled logistics processes ([FSH04]) introduce use cases with communicating logistics elements, e.g. containers. Applications range from first implementations like payload status monitoring (e.g. temperature) to future self-navigation and routing through a network of transportation contractors. An economic handicap of current solutions is the necessity of a dedicated GSM modem for each container, the modem requiring a GSM SIM and associated network operator (sub-)contract (cmp. [BT05]). If the container could instead access an ETC OBU (e.g. over WLAN) as already established transparent gateway to communicate via GSM (or, as it is, an alternative cellular network), acceptance of such an approach to manage and control logistic chains could rise significantly. Again, this would mean extending the logistics use case by ETC aspects.

VAS acceptance issues

Both cases represent applications that have proven hard to establish as stand-alone solutions on dedicated devices and infrastructure. Actually, a number of similar telematics services suffered from lack of acceptance for various reasons ([RG06]). While the details in each case may differ, the main drawback can often be narrowed down to costs. Compared to their apparent economic gain, costs for service deployment or operation were considered too high. Consequently, this resulted in prices potential users – private as well as business – were unwilling to pay.

VAS ETC integration

If introduced into an ETC environment under certain conditions (defined under 3.1.4) however, the services may be operated economically: the ETC system can directly respond to the stated weaknesses of the services by offering to share certain expensive resources. Besides the cellular network access already mentioned above, we find security another demanding aspect. Current state-of-the-art implementations require smart card integration of Secure Access Modules (SAM) for the distributed components, with a corresponding certification authority center-side ([Sc96], chap. 8.12). Cryptographic functions (cmp. 2.3.6) are mandatory for many commercial applications, e.g. to ensure confidentiality and to safely implement payment transactions. Services could benefit significantly if relieved from having to implement a dedicated security environment. ETC-specific interfaces to vehicle data, sensors and antennae are additional potentially rewarding resources to share, either directly as device or on a rather aggregated level as aspect (cmp. 2.3), e.g. as a high-quality positioning component encompassing related functionality like sensor fusion or map-matching.

integration gains

The large number of vehicle-installed OBUs would not only help service operators gain from plain economies of scale. In contrast to proprietary telematics solutions, the ETC domain promotes standardization (cmp. e.g. [RCI07]) and subsequent robustness of interfaces, platforms etc. If the resources can be shared on a therefore stable basis, services add value to the ETC application by helping cover the costs of the ETC infrastructure.

3.1.3.2 Approach II – Automotive Integration

If the above scenario considers sharing ETC resources with value added services, we consequently have to ask the question of an ETC software utilizing non-dedicated hardware. Here also, a number of possibilities arise. Due to the declared automotive focus of this work, we will not yet regard potential alternative run-time environments like those of e.g. smartphones. Instead, we discuss the identification of complementary car electronics modules to provide ETC resources related to the domain discussed in section 2.1.1.

automotive electronics

Modern cars are outfitted with a large number of sensors to gather and evaluate information about the driving profile, e.g. wheel speed, acceleration, yaw for the rotation. As we have seen, with that come run-time processing and memory resources in the form of ECUs.

determining factors for automotive ETC integration

These combined reliability, processing power and communications interfaces – especially of the chassis and multimedia subsystems – represent a potential environment for the implementation of ETC functionality. However, the introduction of additional functions to car electronics has to take a number of determining factors into account.

Proprietary Circulation of Information First of all, while standards for specific bus layers are established (see above), the circulation of the information itself in a vehicle ECU network is encoded proprietarily. Syntax and semantics of data depend on the manufacturer of the component in question as well as the scope of the application implemented by the component. The actual form of e.g. yaw measures of an ESC ECU may differ dramatically from those expected by a navigation sensor fusion. FMS ([FMS05]) is an example for an approach to overcome these differences. In any case, a set of sensor information associated with the vehicle's driving profile that can be gained from the chassis subsystem is potentially valuable input for an ETC application: it can either add to the quality of the tolling process (cmp. 3.1.1.1), or help lowering the costs of operation by superseding dedicated sensors. We can consider the proprietary encoding of information a minor hindrance to ETC integration, as sensor data generally has to be interpreted for ETC purposes, no matter the source.

Specific Characteristics of Busses Depending on the automotive subsystem to integrate with, the ETC components have to connect to the vehicle's communication busses. Especially concerning the quality of service aspect of resource dependencies of the ETC software (cmp. 3.1.1.10), we have to be aware of the diverse characteristics of bus systems: our interactions require conformance to real-time constraints, e.g. the frequency of sensor data transmissions, or the exchange of payment-related messages. Regarding vehicle onboard communications, we find *highspeed real-time control*, *lowspeed* and *multimedia* bus systems (cmp. [ZS07], 1.1). Relevant[5] bus parameters for the given context are

- *message length* and *transmission rate*, defining throughput and influencing the protocol overhead to implement for processing large (composite) objects, e.g. due to the necessity to split messages,

[5]Here we do not need to consider costs explicitly, as an already existing infrastructure would be shared.

- *latency* and *error safety*, related to protocol overhead and real-time constraints of complex transactions, due to potential re-transmissions and timeouts induced by concurrently sharing a bus with a lower priority assigned to ETC, and
- *access assignment strategies*, critical for deployability in a real-time system, strongly determining the risks and efforts related to the addition of ETC processes to an existing onboard network.

busses and ISO OSI layers

Note that the common specifications of CAN, LIN or FlexRay define ISO OSI layers 0 to 2. For transport and application protocols, i.e. layers 3,4 and 5 to 7 respectively, ISO as well as manufacturers like Audi/VW established a number of different and dedicated solutions. Thus, this work has to accept that upper layer protocols are defined by the application's domain and not given universally. Examples are provided by the ISO TP (ISO 15765-2) for CAN, AUTOSAR TP for FlexRay (refer to [AU062]; both layer 4), UDS (*Unified Diagnostics Services*, ISO 14229) and CCP/XCP (*CAN/Extended Calibration Protocol*, European standard ASAM AE MCD 1; all three layer 7). However, it is safe to assume functionality like de-/segmentation of messages, flow control between sender and receiver, and services for communications between application and transport layer. For software integration purposes, C APIs are common for application bus access.

ETC-suitability of busses

While our elaborations will not needlessly exclude an alternative, examining the actual specifications of available busses reveals preferable candidates. Here, for an ETC application a tradeoff is given between the pairs of parameters stated above: the short latency and high reliability of a highspeed C+ class[6] bus like TTCAN (*Time Triggered CAN*, [ISO02]) finds short messages with only a few bytes length. In the case of CAN, the corresponding DATA field has a length of 8 bytes maximum, FlexRay allows 254 bytes. A multimedia bus like MOST on the other hand, provides a general bandwidth of up to 1.2 MB/s for asynchronous communications, but without any guaranteed latency and low reliability due to its non-critical application domain. LIN seems less preferable in the given context, as it was designed as a cheap solution for non-critical low speed applications like controlling headlights, seat and window motors.

bus allocation

Concerning the allocation strategies, we differentiate the methods *centrally/locally controlled*, *random collision free/non collision free* and *event/time triggered* (combinations are feasible). CAN uses *Carrier Sense Multiple Access/Collision Avoidance* (CSMA/CA), an event-driven random approach with a priority-based arbitration phase prior to transmission. FlexRay and TTCAN implement *Time Division Multiple Access* (TDMA), with fixed time slots ensuring exclusive bus access. An advantage of bus control based on time is

[6]Bitrate classification of bus systems defines highspeed classes *C* (125 to 1000 kbit/s), *C+* ($>$ 1 mbit/s) and *Infotainment* ($>$ 10 mbit/s).

predictability: e.g. latency can be determined for any message. Deterministic bus behavior, if achievable, would be complementary to the requirements defined for the ETC software itself in 3.1.4.

Safety Issues Beyond allowing even read-only access to its data busses, an ECU network represents a high-integrity environment (cmp. 2.1.1) and is, as such, generally averse to extension. This is especially relevant for non-native component providers, i.e. third parties other than the vehicle manufacturer and its established suppliers. Consequently, any element of the network – hardware and software – has to submit itself to the mandatory defined requirements of the environment: specifications of interfaces, resource consumption and access, structure and behavior. Self-diagnosis and -monitoring are mandatory safety measures during run-time. Naturally, strictness of these requirements rises with the criticality of the respective subsystem. They may be comparably modest in the multimedia context, but get unforgiving when we cross the threshold to potentially endangering passenger safety, e.g. concerning safety measures of the chassis. [IEC05] describes requirements for a quantitative proof of functional safety for electrical and/or electronic and/or programmable electronic (E/E/PE) components that is suited for its subtype automotive ECU. Risk-freedom of system or equipment operations is brought about by a risk-based approach of a complete safety lifecycle covering all development and operational activities from initial concept to decommissioning. With specified techniques and measures the *safety integrity level* (SIL; 1 to 4, with SIL4 being the highest) of a system may be determined.

Interactivity Beyond the Vehicle A further obstacle to automotive ETC integration concerns the system's interactivity beyond the vehicle itself. For the safety-critical chassis and power train subsystems we find test, diagnostics and maintenance interfaces in the form of *gateways* (cmp. [MS07], 2.2.3). Gateways provide indirect access to onboard electronics, a uniform interface and optimized bus load by isolating diagnostic from regular information streams. The associated ECUs should be accessible exclusively for workshops licensed by the manufacturer, although unwarranted flashing ("chip tuning") abounds and has to be countered, e.g. by trusted flashing ([SRM06]) to achieve tamper-proofness. In- and outbound communications of the moving vehicle in a state of operations beyond maintenance are restricted to information and entertainment subsystems. Introducing a highly interactive component like an ETC software into the automotive environment in general, without strictly limiting its access, would again imply complete conformance of the new component to automotive requirements including security and monitoring. The existence of gateways, however, represents a potential integration approach.

Run-time Environments Finally, any automotive software integration endeavour has to cope with the lack of standardized architectures and run-time environments as stated in 2.1.1. While established methodical standards exist for the specification, implementation and testing of ECU software (cmp. 3.2.2), acceptance of automotive application frameworks suffers from a number of problems (also cmp. [ZS07], 7.1).

- *Initiatives and standardization comittees consist of competitors*: whenever a universal solution is about to emerge, interests of the participants collide. Proprietary advantages are protected and not put forth, potential benefits for other parties forestalled.
- *Overlapping initiative participation*: in order to observe any development and (potential) competitors, commitment to a single standard is seldomly found, especially within large companies.
- *Integration with existing proprietary technology*: No manufacturer is willing to migrate an operational, complete automotive electronics platform just to conform to a new framework. Also, concurrently to the standardization activities, actual development proceeds. Premature products preempt the standard, forcing it to factor in their characteristics and features, often plainly due to the weight of the corresponding player.
- *Internal development standards*: Especially in the context of high-integrity systems and liability risks, established and certified development methods are strictly preserved. If they collide with requirements of the general standard, it is modified – resulting in a proprietary version.

As a result, we find "conformant" or "compatible" solutions, applications, modules, tools, which are actually exhibiting discrepant traits. Similar examples[7] can be found with *Offene Systeme für die Elektronik im Kraftfahrzeug / Vehicle Distributed eXecutive* (OSEK/VDX, [OS05]), *Herstellerinitiative Software* (HIS, [HI04]) and *Automotive Open Systems Architecture* (AUTOSAR, [AU06]).

conclusion of the above issues

These problems cannot be solved solely by technical means, however efficient they may be. In consequence, it could be detrimental to our approach if it would select and directly build on one of the aspiring standards, or would be too tightly integrated with its specific workings. Instead, the question for the underlying shared aims and features of the automotive frameworks arises. There, we find the objective of realizing

- *highly dependent* (e.g. in [OS042]),
- *concurrently running* (e.g. in [HI03])

[7]Each of them presenting intersecting participants like BMW, Daimler, VW etc.

- *real-time components* (e.g. in [AU06])
- in the *restrictive* ECU environment,
- exchanging *messages* based on *communication layers* (IO Library in HIS, OSEK COM in OSEK/VDX, Virtual Function Bus in AUTOSAR) abstracting the automotive busses.

Thus, potential standard adherence in the context of this work would primarily manifest itself in conformance to the respective interface specifications. In order to at least alleviate integration into any such future framework, our architecture should complement, not contradict, their fundamental requirements and constraints, like [AU063]. Furthermore, as discussed in 2.1.1, below and 3.1.3.4, there is good reason to disregard third-party components in the given ETC context due to operational risks and the desirable high component controllability.

a new telematics subsystem

Potential Integration Approach Corresponding to the differentiation of the domains in chapter 2 and the previous descriptions, a potential automotive ETC integration scenario justifies a telematics subsystem (cmp. fig. 3.15), complementing the established subsystems as described in 2.1.1.

why new

On first glance, the multimedia subsystem seems to support ETC requirements sufficiently in order to justify the aggregation of ETC components there. As was shown above however, some crucial aspects cannot be provided, as guaranteed reliability and high integrity of commercial ETC transactions are less of an issue in the context of infotainment, navigation and communications. Characteristics of e.g. the MOST bus are not congruent with the real-time ETC component interactions. And while real-time constraints are satisfiable in the chassis subsystem, its transformational signal processing capabilities, comparably low bus bandwidth and car safety criticality are inapplicable to the direct integration of reactive, interactive ETC processes.

how to (HW)

This leaves the connection of dedicated ETC ECUs to a bus like TTCAN or FlexRay (see above), that is in turn attached via subsystem gateway to the vehicle network's central gateway. A smart card SAM and DSRC module are introduced into the automotive environment by telematics, or more specific, ETC use cases. Once installed in the vehicle, a secure access module could be utilized by other subsystems, e.g. for trusted flashing or general access control to car electronics, increasing the economic attractiveness of this scenario. As pivotal element, the primary ETC software with business logics and processes is implemented on a sufficiently capable[8] micro controller.

how to (SW)

Once this telematics subsystem thusly participates in the vehicle ECU

[8]The actual capabilities – processor run-time, memory – are determined by specifics of the ETC use cases: necessary computations of the selected locating algorithm in MIPS, real-time concurrency constraints concerning payment processing, enforcement and so on.

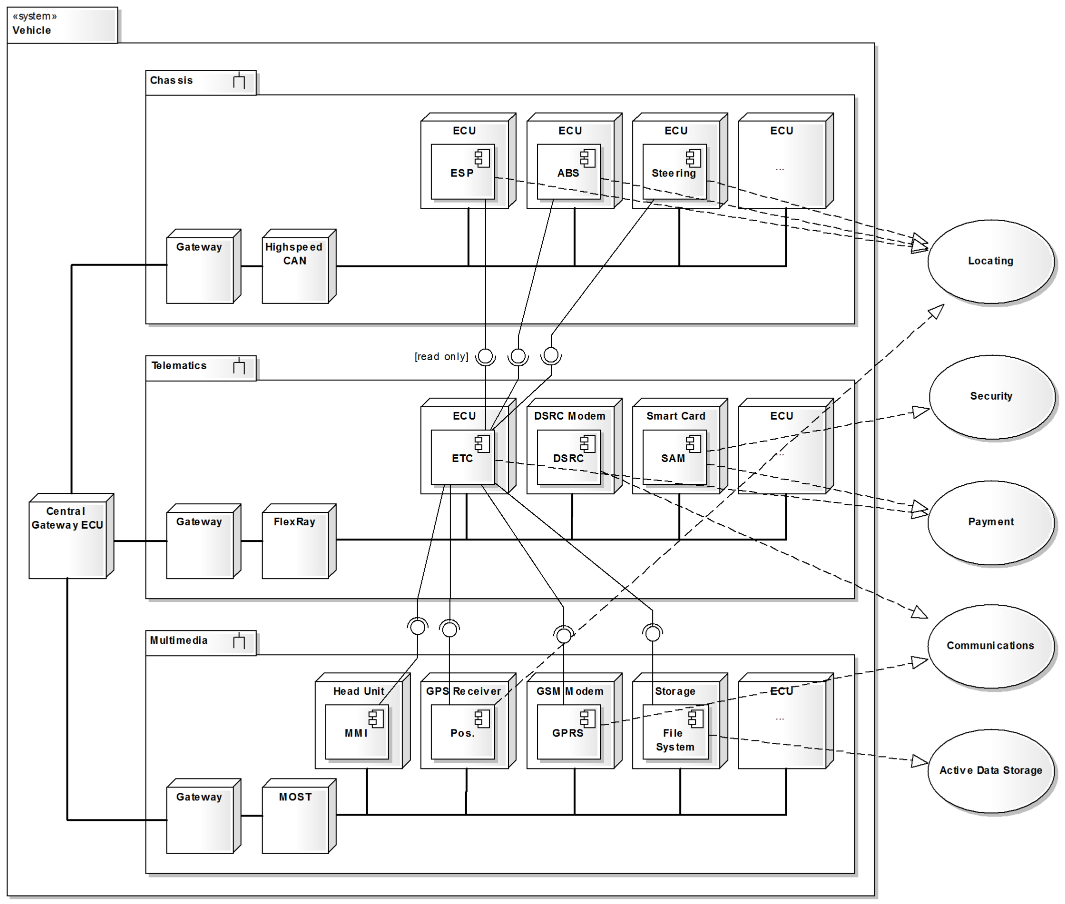


Figure 3.15: Automotive ETC Integration Scenario

network, interfacing the software to selected ECUs of the other subsystems at least partially implements ETC aspects (cmp. 2.3; fig. 3.15 illustrates this mapping) without the need for additional dedicated resources. Concerning locating, the ETC application may acquire a position from a navigation system's GPS receiver, improving it with data gained from chassis sensors[9]. To avoid compromising the integrity of the critical chassis subsystem, the interface is strictly read-only. This may be realized e.g. by the corresponding ECUs just writing every *n*th sensor value to their gateway, where other subsystems can fetch it via the central gateway (e.g. diagnostics interface) without depending on internal chassis access. If the ETC ECU memory itself can handle critical scenarios (e.g. concerning abrupt current interruption), static storage of

[9]It is questionable to merely use the locating result of an arbitrary navigation system, as ETC may require a dedicated, reproducible locating algorithm due to certification and liability reasons.

the multimedia subsystem may provide data persistency for less critical cases. The GSM module of a car mobile phone installation can grant cellular network access to the ETC processes, realizing a non-time-critical part of the communications aspect.

utilizing other automotive resources in ETC processes

Here, we may derive a general rule for the utilization of non-dedicated resources in the vehicle: units involved in time-critical transactions (like the SAM in payment or DSRC in the enforcement) have to be included in the telematics subsystem (real-time bus with sufficient bandwidth), together with the ECU running the ETC software. Others, with more relaxed real-time constraints allowing a certain lag (e.g. positioning), or which implement services from which non-deterministic behavior is expected (like GSM services), may be accessed by gateways and connected to other busses.

acceptance

We are well aware of the fact that given the current automotive systemscape, the above approach today might be considered too risky. Especially component suppliers need to protect their product's domains from intrusions of third party elements due to issues of liability. Nevertheless, the situation is similar to the introduction of performant bus systems and programmable micro controllers into cars in the early 1990s (cmp. [SZ06], 1.2). The consequent departure from autonomous, specialized controllers leads to the realization of more economical high-level software functions sharing e.g. sensors without the need for dedicated wiring. With the ETC scenario, large-scale deployment and cost pressure will eventually open vehicle electronics for integration. Especially if there is additional business involved for car technology suppliers, which are already implementing very similar inter-subsystem collaborations – albeit still proprietary, as was shown in chapter 2.1.1.

3.1.3.3 Approach III – Interoperability

The introduction to this work implied the assumption that international road tolling systems will always represent a heterogenous systemscape, even if this might sometimes manifest itself only in partial aspects, like the tolling scheme or tariff model. A number of practical reasons – e.g. the limited facilities for OBE fitting in a vehicle – call for the interoperability of ETC installations. Already confirmed and corroborated by the EU (cmp. directive [EU04] with corrigendum [EU042], [RCI07]), interoperability of a larger number of ETC systems furthers economy. Again, achievable by sharing available infrastructure resources[10], this time between a number of ETC operators.

interoperability requirements

Interoperating ETC requires a vehicle's OBE to transact with a multitude of operators respectively service providers. For that, a number of preconditions have to be met: the corresponding devices and interfaces have to be present, operating and configuration data (called *toll context data* in [RCI072], 2.2)

[10]The fact that a corresponding harmonization of interfaces also helps to create an open market for ETC OBE, again promoting economy, is out of the focus of this work.

up to date for each operator and the OBU software has to implement the respective operator's business processes – specific locating, tolling computation, transaction protocols.

focus on the ETC software

Most of the OBE hardware infrastructure in question may be considered universal for realization of the aspects of any GNSS/GPS ETC system. Deployment of proprietary modules is very unlikely due to high production and replacement costs. Apparently, the hardware, with one possible exception, is fixedly installed and thus actual change of modules is associated with high effort. With this precondition of compatible hardware, interoperability therefore almost exclusively pertains to the OBE software.

realization scenarios

Here, we find three general realization alternatives:

Multi-operator software – one application implements the operational transaction sequences and interfaces of multiple operators. For this purpose, it aggregates specific libraries and/or threads, switching the internal context initiated by activation events like entering an operator's area (GPS) or an explicit signal by roadside infrastructure (automatically) or the user (manually).
Advantages: Unified handling of multiple processes, one software to manage, update and maintain.
Drawbacks: At least limited need for disclosure of proprietary algorithms by each operator. With a growing number of operators to integrate, software may get too complex for the run-time environment.

Multi-application OBE – the OBU runs a multitude of independent applications concurrently. Each process is responsible for implementing its respective operator's business logics, activation and dormant states.
Advantages: Clear assignments of responsibilities on the business process level, encapsulation of proprietary activities.
Drawbacks: Requires complex real-time OS to provide safe mechanisms for concurrency management. Costly testing and integration due to the lack of a centralized management component. Run-time environment limits the number of operators on the OBE (memory, file handles, processing).

Application switching – a framework manages updates, installation, activation and removal of ETC application modules during run-time. These ETC operator software components implement a corresponding interface, but may keep their internal structures opaque.
Advantages: Unified management of multiple operators. Generally unlimited number of supported operators due to the ability to replace components dynamically.
Drawbacks: While the process leading to activation may be complex, the framework nevertheless has to guarantee timeliness. Elaborate frameworks allocate scarce OBE run-time resources.

security as determining aspect

In any of the interoperability scenarios, we find security a most critical aspect. As stated in 2.3.6, it is an integral part of all processes of the ETC software. In addition to that, it represents a part of the system the operator is very unlikely allowed to delegate or disclose lightly (liability). Integrity of the transactions is a prerequisite for commercial and legal validity. Consequently, distributed elements of OBE security, like algorithms for en-/decryption, signatures, are encapsuled in smart cards, providing an environment with strictly controlled access (cmp. [RE08], chapter 16).

separation of generic and operator-specific parts

The fact that these SAMs are – compared to the other installed equipment – OBE hardware components that are in their chip card manifestation designed for easy replacement[11], indicates an approach for interoperability: we suggest a systematic separation of the universal OBE with a generic resource management software from a completing operator-specific business process implementation on a smart card, in addition to its cryptographic functions. This offers a robust variant of application switching by physically exchanging the card in conformance with the travelled operator domain.

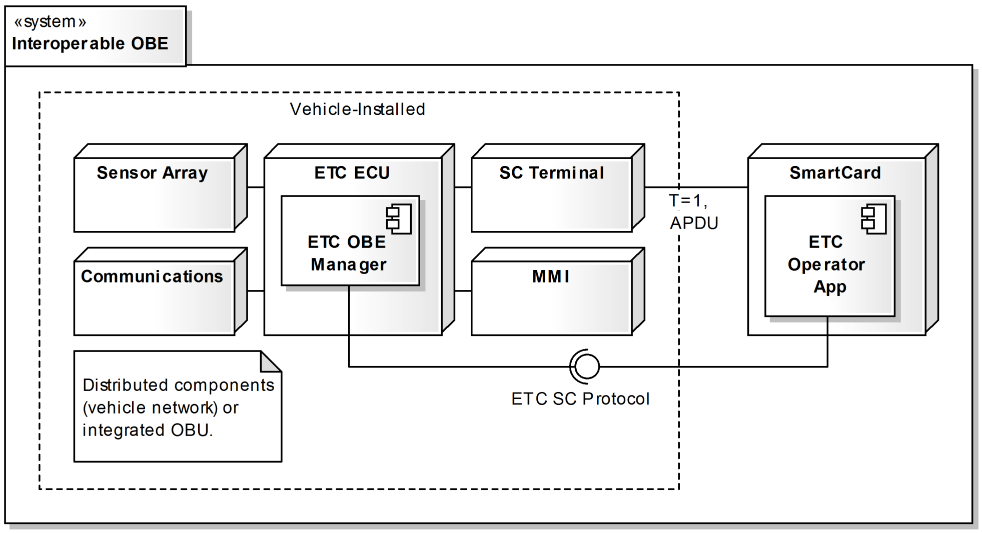


Figure 3.16: Smart Card Integrated Interoperability Scenario

deployment scenario with OMS

Figure 3.16 illustrates the approach. The *ETC OBE Management Software* (OMS) acts as middleware or interpreting agent between the smart card ETC application and OBE resources: it implements interfaces to communications, positioning, data storage and user input/output. Note that this scenario may coincide with the previous approaches 3.1.3.1 and 3.1.3.2. In these cases, the

[11]Not possible in the current deployment of the German system, as the smart card reader is inaccessible without opening the OBU. This, however, is installation-specific and does not constrict the general replaceability option.

OMS additionally needs to factor in concurrently sharing OBE modules. The OMS is effectively controlled by the smart card logics. Its application defines the actual ETC software activities conforming to the specifications of the corresponding operator – transactions, states, actions, incoming and outgoing messages, their frequency and conditions. We suggest using the asynchronous, block-oriented, half-duplex *T=1* transmission protocol (cmp. [ISO97]) between terminal and card, as it allows for secure messaging.

Note: A related but very vague approach was patented by [DC03], in that a smart card computation unit shall be used for the identification of toll road segments, a number of plausibility checks and security, interacting with some vehicle-fitted hardware in an unspecified way. However, [DC03] completely disregards the determining, critical realization issues and thusly renders its description effectively as obvious as arbitrary: a) The smart card is explicitly required to handle the computation necessary to identify a road segment from a digital map section and a set of positions. The card hardware may not be able to process the necessary computation in time or at all. No alternative mode is provided for this likely[12] scenario. b) The smart card is explicitly required to actively, initiatively control its automotive environment. As is widely known (e.g. from the ISO 7816 standard applying here) and described below, this is not feasible directly. Instead, while not overly complicated (cmp. e.g. *SIM Toolkit*, [ETSI96]), the relevant question would be how to represent the desired, ETC domain-specific interaction scheme with the means actually provided by the smart card. No answers whatsoever are given by [DC03].

smart card message specifics

The characteristics of interaction between ETC OBE management and operator process implementation depend on the capabilities of the deployed smart card. To begin with, messages have to be resolved into *Application Protocol Data Units* (APDUs; [ISO05]). Current smart card communications conform to a master-slave relation, with the terminal issuing commands as *C-APDUs/command APDUs* and the card responding with *R-APDUs/response APDUs*. Length of the unit's *Data* field is specified in the fields *Lc* (*Length command*) and *Le* (*Length expected*), either encoded in a byte, resulting in a maximum length of 256 bytes, or, in the extended case, up to three bytes, resulting in 65,536 bytes maximum.

overcoming master-slave

In consequence, the smart card ETC software may not actively control the OMS, as it is unable to initiate an interaction. Instead, here we suggest a protocol implying a strictly reactive ETC application that expects specified messages from the OBE, processes these and responds with commands to the management module. We find a similar mechanism with the *SIM Application Toolkit* (SAT, [ETSI96], [3G03]). That solution realizes *proactive* card commands by coupling the smart card application with a mobile device counterpart continuously polling the card for function call messages. In the case of SAT, the *STATUS* command is used for that purpose commonly every 20

[12]Based on current road segment identification algorithms and smart card capabilities.

seconds, as it is short and quick to transmit (cmp. [Ra06], 5.8).

polling with payload

In the ETC scenario, liveliness of this interaction is perpetuated by the regular and frequent stimulus of positioning, inherent to the ETC domain and its use cases (cmp. chapter 2). The payload-free polling message – resembling a *busy waiting* – can therefore be substituted by the position updates, required by the modules implementing the locating aspect.

regarding card limitations

Furthermore, like in 3.1.3.2, we have to take the limited run-time and memory resources of a smart card into account. Depending on the operator's available budget, the ETC software might have to work on a 16 bit processor and with as little as 64 kb RAM[13]. As a variety of cards may be deployed by different operators, the OMS has to integrate them flexibly, according to their respective performance profile. This is also congruent with a common mechanism of the smart card domain: the initial exchange between terminal and card sees the *reset* and the card-specific *Answer To Reset* (ATR, [RE08], 8.1), declaring data transmission and chip card characteristics to apply to the subsequent transactions.

ETCSC configuration

The ETC scenario complements these predefinitions between terminal and card with domain-specific predefinitions between OMS and ETC software. Depending on the card hardware the ETC application is running on, this approach has to be taken one step further, beyond the agreement on communications parameters. Therefore, an ETCSC (*ETC Smart Card*) configuration consists of the entries depicted in table 3.1. The definition includes constraints and quality of service requirements to fulfill by the OMS when providing data or accepting commands, e.g. value array sizes[14], time cycles and frequencies, maximum durations.

computation distribution

The *external computation instructions* represent most critical entries in this context. Corresponding to its individual run-time capabilities, a card may partially delegate its ETC application's workload to the OBE, e.g. in the case of computationally intensive algorithms. This way, the ETC business process can be realized by the overall, integrated OMS/ETC card software system, even if the smart card hardware cannot handle it completely.

the Java Card view

The ETC domain requires different features for sharing on-card and off-card resources than e.g. provided by the Java Card platform ([Sun061], [Sun062], [Sun063]). Its *Remote Method Invocation* (RMI) service complies to the master-slave principle, denying the card applet access to methods on the terminal/-Card Acceptance Device (cmp. [Sun061], 8.1.1.3). Via the package *javacardx.external* (cmp. [Sun063]), the applet may only be enabled to access external memory subsystems.

ETC-specific view

However, even the ability to actively invoke remote procedures of the OMS might not be sufficient for the given scenario. This would imply predefinition

[13]While there are cheaper cards with 8 bit processors addressing 6 to 30 kb on the lower end of the performance spectrum, we will consider them as clearly unfit for the ETC scenario.

[14]E.g. for the case of having to pass more than one position at once to optimize the number of exchanged messages in a given time.

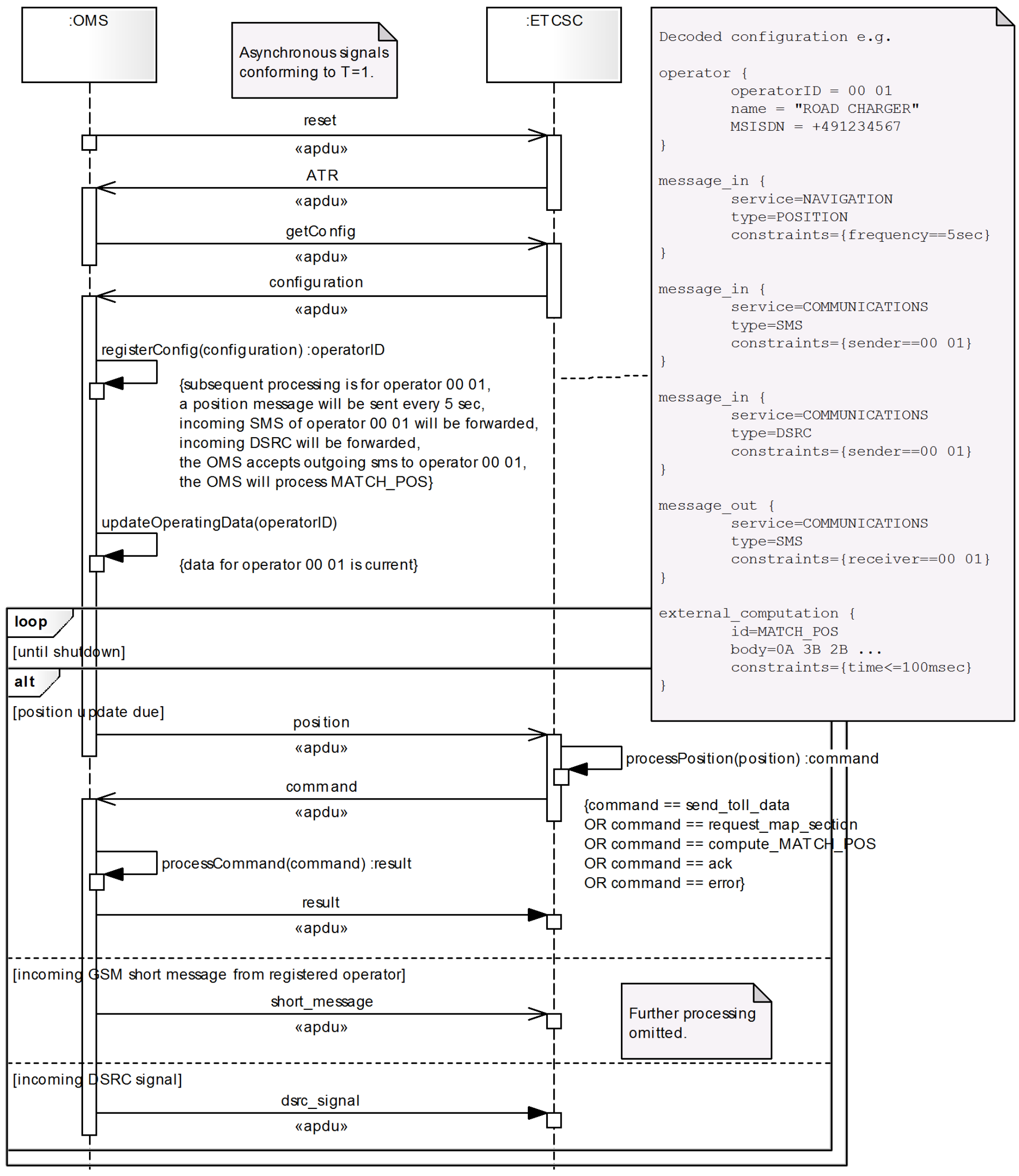
Entry (Cardinality)	**Generic Record Format**	**Description**
ETC Operator (1)	`[id][name]`	Unique ID required for associating incoming and outgoing messages, operating data management, payment record accounting; name as string for HMI display.
Incoming Message Registration (*)	`[service]` `[type]` `[constraints]`	Information received or generated by the OBE to be passed on to the ETCSC, with defined quality of service.
Outgoing Message Registration (*)	`[service]` `[type]` `[constraints]`	Messages the ETCSC software will be sending and is expecting to be accepted, processed and/or passed on by the OBE with defined quality of service.
External Computation Instruction (*)	`[id][body]` `[constraints]`	Computation delegated by the ETCSC to the OBE to be processed in conformance to defined constraints.

Table 3.1: Generic ETCSC Configuration

of any operator-required functions on the OBE/OMS, contradicting a strict separation of a universal OBE and specific card, and in turn requiring the operator to disclose specifications. Instead, the OMS has to provide generic computation functions, e.g. for complex mathematical operations, as well as an interpreter for operator-specific computation instructions stored on, passed by and on completion returned to the ETCSC. Depending on the actual ETC algorithms, this mechanism allows for the time-optimized distribution of workload. We will revisit it in chapters 4 and 5.

interaction example

Figure 3.17 illustrates an example of the above OMS/smart card ETC application coupling. After exchange of `reset` and `ATR`, the OMS requests the `configuration` as APDUs and registers it upon receival. A successful registration implies a number of postconditions. The current tolling operator with the ID 00 01 is selected for interactions with center systems and accounting. The OMS is set to feed the ETCSC application position data every five seconds, generated by some OBE navigation service, and incoming SMS and DSRC messages from operator 00 01, received by a communications service. It expects short messages from the ETCSC software, which to forward to operator 00 01 via an OBE communications service. Finally, the OMS will accept computation instructions identified by MATCH_POS and described by a binary


:OMS
Asynchronous signals conforming to T=1.
:ETCSC
reset
«apdu»
ATR
«apdu»
getConfig
«apdu»
configuration
«apdu»
registerConfig(configuration) :operatorID
{subsequent processing is for operator 00 01,
a position message will be sent every 5 sec,
incoming SMS of operator 00 01 will be forwarded,
incoming DSRC will be forwarded,
the OMS accepts outgoing sms to operator 00 01,
the OMS will process MATCH_POS}
updateOperatingData(operatorID)
{data for operator 00 01 is current}
loop
[until shutdown]
alt
[position update due]
position
«apdu»
processPosition(position) :command
command
«apdu»
{command == send_toll_data
OR command == request_map_section
OR command == compute_MATCH_POS
OR command == ack
OR command == error}
processCommand(command) :result
result
«apdu»
[incoming GSM short message from registered operator]
short_message
«apdu»
Further processing omitted.
[incoming DSRC signal]
dsrc_signal
«apdu»

```
Decoded configuration e.g.

operator {
        operatorID = 00 01
        name = "ROAD CHARGER"
        MSISDN = +491234567
}

message_in {
        service=NAVIGATION
        type=POSITION
        constraints={frequency==5sec}
}

message_in {
        service=COMMUNICATIONS
        type=SMS
        constraints={sender==00 01}
}

message_in {
        service=COMMUNICATIONS
        type=DSRC
        constraints={sender==00 01}
}

message_out {
        service=COMMUNICATIONS
        type=SMS
        constraints={receiver==00 01}
}

external_computation {
        id=MATCH_POS
        body=0A 3B 2B ...
        constraints={time<=100msec}
}
```

Figure 3.17: OMS and ETC Smart Card Interaction Example

body, which to process in 100 msec maximum after invocation.

why validate on insertion

While it is generally possible to do without the explicit declaration of required services and assignment of computations, and handle these dynamically on invocation, there is a practical reason for this configuration step. The user is expected to insert the card prior to using the operator's traffic infrastructure. On insertion, the OBE may validate the ETC software's requirements and check for conformance. If a problem should arise, the user may be warned accordingly and thusly avoid being singled out by the enforcement (cmp. 3.1.1).

example continued

The operating data associated with the processes of 00 01 is updated. After that, the main loop is entered and iterated until system shutdown. With a frequency of five seconds, a position update is due and sent, encoded in an APDU, to the ETCSC application. The position message is processed, resulting in an updated ETCSC application state and a command to the OMS: the APDU may encode a request to forward accumulated tolling accounts to the center systems, a request for updated map data, a request to match positions to the map, or a plain acknowledgement or error. This command in turn is processed by the OMS, which then replies with the result. Between the position updates, incoming short and DSRC messages are forwarded and processed correspondingly.

3.1.3.4 General Angle

general economy aspect

A more general aspect of economy pervades the three previous approaches. Concurrent operations in an embedded environment with limited processing and memory resources call for lean, resource-aware implementation. Features of run-time engines should be reduced to the essentials.

influence of "standard" solutions

However, our professional experience showed that the introduction of Java, .NET/Windows mobile and other "standard" run-time environments, frameworks and components in cadence with the capability upgrading of mobile phones and Personal Digital Assistants (PDAs) lead to a similar attitude as can be found in the desktop and server domains: memory and processing power are not an issue anymore; they are abundant.

consequences ...

This is obviously a questionable approach for the still restrictive platforms ECU and smart card described above. But even if, given time, these restrictions would be removed by advancements of the respective hardware technology, deployment of standard run-time engines or components imply consequences for reliability and economy.

... of deploying third-party components

Complex third-party products like the named may reduce net implementation costs. On the other hand, controlling the risks they introduce into the system is expensive: either the component is certified (cmp. e.g. section 3.2.2), which is associated with a higher price, or it has to be tested as black box with high efforts. Furthermore, nonessential but integrated features may use up valuable system resources, while stripping the component down may be either restricted (no source code available) or costly due to the necessity for the developers to become deeply acquainted with it.

... of developing dedicated components

In the case of a high-integrity system, developing a specialized run-time environment and application is, on a first glimpse, more costly than buying standard components and just putting them together. But the other side of the balance sheet finds optimized resource usage due to systematic customization, lowering the price of hardware or at least leaving resources for other applications. Explicit and systematic controllability (cmp. requirements I to III in section 3.1.2) leads to more efficient tests and thusly lowered risks.

3.1.4 Economy-driven Requirements

All of the discussed approaches to an economically operatable ETC software focus on resource sharing. Thus, definition of the requirements concerning economy has to ensure that conformance is equivalent to the prerequisites for stable concurrent operations in the given domain context and technical environment.

component requirement

Approach I aims at the safe operation of VAS components: VA service processes must not have any effect on ETC processes whatsoever, and ETC processes have priority in any given situation[15] , i.e. system state. A sensible consequence is to isolate VAS from ETC functionality on the implementation level. If the VAS accesses ETC resources, it has to use a defined, controlable interface. The need for a central controlling element has already been expressed by requirement I. So, we postulate the necessary structuring with

> *Requirement IV (component organization): any interactive, non purely transforming process (designated a service) has to be implemented in the form of a component. The component completely encapsules the service behavior and provides message queues for synchronous or asynchronous input and output.*

This requirement has implications for an implementation in the context of this work. As a departure from the functional, transforming ECU applications common to the automotive domain (cmp. 2.1.1), we have the *service*. According to requirement II, its behavior has to be defined by a state automaton. The service reacts to signals in the form of messages that we allow to take an arbitrarily complex form, from records to chained messages conforming to a protocol (cmp. e.g. [Do04], chap. 3.4.7). In contrast to the classic stateless electronics control circuit with directors, sensors and actuators based on plain numeric values, we therefore get a network of communicating state machines, less restrained in their interactions. If we further apply requirement III to the known component approach (cmp. e.g. [Do04], chap. 2.4.3), a differentiation from the common concept of hidden internals and public interfaces becomes apparent: internal behavior of the component implementations is not allowed to be opaque. Instead, the controlling instance proclaimed by requirement I a) keeps track of every component activity and b) orchestrates the component interaction, and for that purpose has to provide and schedule a message bus. It is left to the architecture foundation in chapter 4 to incorporate this feature

[15]With the possible exception of an emergency call. But this special case has no influence on the general approach and the need for priorities.

in a non-intrusive way so as not to obstruct maintainability and replaceability of components.

Approach II sets the ETC application into an automotive environment to safely coexist with vehicle electronics functions and concurrently access their resources. As discussed, depending on the criticality of the respective subsystem, the ETC software has to conform to the defined standards enforcing high-integrity of installed components. We pursue a general acceptance of a new, behaviorally complex component that is frequently interacting with outside systems over various interfaces like GSM and DSRC into a hitherto restricted system. Consequently, preemptive support of any potential certification concerning software stability seems to be in order. Deterministic and controllable behavior is aided by requirements I to IV, which will reflect in the static and dynamic architecture. The conformity to reliability standards and guidelines (for an overview refer to [Sto96], chap. 14) however, also strongly concerns the implementation.

statics requirement

From a number of sources – [MI04], [Ba03], [BDV03], [Sto96], chap. 9, [ISO00] – we can derive generalized characteristics of a software considered highly reliable. Based on these, we define a set of high-integrity requirements for the ETC software.

Requirement V (statically determinable implementation): an implementation of the architecture shall avoid dynamic structures and mechanisms. No dynamic memory allocation during run-time, including class instantiation, is allowed. Implicit allocation is allowed only if bounded and the maximum size is determinable during compile time. Types of all values of assignments and operations have to be determinable during compile time; no dynamic bindings, polymorphic functions or data types are allowed.

This satisfies a very basic condition for high-integrity systems: deterministic memory consumption of the application is fundamental for reliability. If the memory footprint of the ETC software is specified for a given configuration of service components, hardware resources may be selected accordingly, and out of memory run-time errors due to unbounded storage are precluded. We will allow recursion in a controlled way, so that the resulting implicit stack memory allocation can be measured prior to run-time, e.g. by examining the maximum depth of recursive function calls. Further potential run-time problems induced by type incompatibilities, e.g. in implicit conversions of polymorphic operations, are precluded by a general directive for static typing. This especially forbids assignments of subclass instances to variables of their parent classes in any context, which – as we will see in chapter 4 – directly influences the freedom of architecture design.

The cited standards and guidelines define a range of additional rules for reliable systems. They may be applied to the implementation of an ETC software, but have no further impact on the elaborations of this work[16].

determinism requirement

After the obligation for deterministic memory usage and typing, we consider the run-time behavior with

> *Requirement VI (deterministic run-time behavior): a software based on the architecture shall exhibit deterministic run-time behavior concerning concurrency and scheduling.*

This complements requirement V: implied is e.g. a fixed number of processes with static priorities to facilitate timing analysis (aimed at a potential certification). By denying the application complex, dynamic concurrency features like arbitrarily adding or removing threads, also the requisites concerning the run-time environment are kept to a minimum. In turn, costs of its hardware platform are reduced (aimed at economy).

persistency requirement

The final requirement is the implication of a domain-specific scenario, which strongly impacts a certain aspect of the architecture:

> *Requirement VII (persistency of activities): the complete system configuration of requirement III has to be persistent for selected points of time t.*

A necessity due to the fact that a large number of trucks is registered as transport for hazardous goods. On switching off the ignition, electronics of these vehicles have to shut down immediately. In practice, we find a shutdown time of approximately 20 milliseconds, after which the electric current is interrupted (known as *clamp 30*). As the continuity of process chains is critical for ETC functionality – from locating to payment including data update protocols – measures have to be taken to ensure continuous processing even under unstable conditions. Preserving the full configuration of the system after reaching defined process states provides a fallback position for power loss and events with similar effects. But we also find justification of this requirement beyond the vehicle domain: Running an ETC application on e.g. smart phone platforms removes the immediate physical hazards for third parties, but introduces risks for the valid operation of the ETC business processes. A common smart phone OS is generally capable of buffering a full system state on shutdown. However, to protect the legally relevant data of our software, it is necessary not to rely on other party functions; instead providing a self-managed solution.

This concludes the definition of requirements for a reliable and economical ETC architecture. By conforming to IV to VII, the ETC software can be operated economically and safely in a shared environment. It may take the form

[16]Many of them can be regarded as "common sense" today (e.g. no use of *gotos*), anyway.

of an ETC platform with additional services, a smartphone, or an automotive ECU network running an ETC component.

3.2 Concerning the Implementation

It is a well established practice to abstract the model from the actual system implementation. The design may then focus on essentials and does not have to concern itself with technical details, often regarded as trivial[17]. In the given case however, early discussion of the issue of implementing the ETC software is necessary, while keeping an abstraction from the programming generally in effect.

3.2.1 Model-Program Interdependencies

interdependencies between model and program characteristics

Due to the previously defined requirements for the architecture, a set of programming characteristics has to reflect in the system's model and its design. Early consideration of these implications becomes especially important as they mostly constrict the use of common modeling patterns and their approaches, like polymorphy: in the given context, a buffer, for example, may not be defined generically to handle instances of any subclass of some generalized *Message* class. To ensure conformance of the subsequent implementation to the explicit and implicit design constraints given by the requirements, first, the software model has to incorporate them – in order to actually be implementable in the required way.

implicit reliability requirements

Furthermore, only certain constituents of the referenced guidelines and standards were treated explicitly in the previous section, like determinism. By additionally providing sufficiently defined semantics, strong data typing, and facilities to protect against memory exhaustion at run-time (cmp. [CGW91]), a language can immensely benefit the reliability certification of a product based on the architecture introduced in this work. Naturally, a bad choice can have the reverse effect, up to even completely denying the software a desired certificate. Aspiring to comply to those general measures commonly considered to achieve reliability in a software is an implicit requirement complementing the definitions of sections 3.1.2 and 3.1.4. And again, the model has to ensure that conformance is not hindered by its design.

scrutinizing an established rule

We are aware that the consideration of implementation details on the model abstraction level may seem presumptuous and seems to question the apparent. Nevertheless, the unguarded, unreflected exercise of this established rule may lead to problems in practice. Consider a potential counterexample to the rule in [Ko97], chap. 4.1.3: Without reservation it declares data representation (syntax) as irrelevant for real-time system models, suggesting "gateway components" to take care of differences between subsystems. This implies an

[17] And rashly so.

unwarranted disregard for implementation restrictions that are quite common especially for embedded RT applications; restrictions that might impede the encoding of certain data value semantics, e.g. regarding their precision, or restrictions of interface realizations that forbid the necessary transformations due to limited bandwidth. An actual example from the ETC domain is given by GPS positions and their different representations concerning allowed memory usage in the receiver and map data storage, which must be respected in the model design.

other approaches

For similar reasons we completely dismiss the "Executable UML" approach of [MB02] for the ETC software domain. The abstraction layer upheld by the action semantics approach given there forces an independence from the software platform and especially the programming that has to be considered directly opposed to the transparency requirements and liability of the ETC software architecture (cmp. [MB02], p. 110).

3.2.2 Language Selection

Consequently, the selection of a programming language has to consider a) the general fitness for a high-integrity context and b) the specific requirements from 3.1.4, namely V and VI. In both cases, the language itself as well as its run-time engine[18] (or kernel) are subject to examination. This is owed to the fact that we cannot necessarily presume availability of an operating system in our architecture, e.g. in the case that the software has to operate on a smaller system like an ECU.

selection criteria

Criteria for programming languages in high-integrity systems were discussed in [Sto96], chapter 9. Besides the ones already mentioned, we find the questions for simple, formal definitions of language features and the detection of violation of language definitions during compile time. [RE92] implies general requirements of analyzability and predictability. A comparison in [CGW91] revealed Modula-2 as relatively best suited for the field in question, in that case compared to e.g. C, Ada, Pascal and Assembler. However, the result also revealed a problem this work is confronted with 16 years later: there are factors outside the rather technical scope of such an analysis that can lead to disqualification. Limited industrial acceptance, either resulting from or leading to insufficient support by vendors of development environments and lack of qualified personnel, effectively rules out the respective language.

Java

At the time of this work, the relevant domains of telecommunications, telematics and automotive see C, C++ and Java as preeminent for many applications (cmp. e.g. [Sc07]). Although Java tries to invade the domain of automotive telematics for some time now, the associated development is usually about non-critical, informing systems and car-multimedia. We are aware

[18] This work understands the run-time engine as provider of – besides basic functionality, cmp. [KPRR91] chap. 2.6 – memory management and concurrency features (threads, scheduling, synchronization, timing etc.) for the application.

of a number of activities to emphasize safety aspects in Java (e.g. in the smart card sector, Ravenscar Java [KWK02]). Anyway, most of these are either strictly proprietary and/or still in a beta or prototype stage. In a recent analysis, [BW06] also came to the conclusion that many of Java's principles collide with the issues of safety-critical systems: due to its OO characteristics and error-prone lexical structure it is hard to define a safe language subset; subsetting for e.g. static analyzability conflicts with the general dynamic approach. [BW09] still find it appropriate for dynamic, soft real-time applications, but raise concerns over RTE efficiency (cmp. chap. 16 there).

C/C++

On a first glance, this leaves C/C++. Pertaining to the deployment on an automotive platform, any development endeavour is confronted with established approaches. Most prominently, we find MISRA C [MI04] with the development environment dominated by the Mathworks suite ([Ma07], with Matlab/Simulink and Stateflow). HIS ([HI04]) as well as the Run-Time Environment of AUTOSAR, for example, require their code to conform to the MISRA C standard (cmp. [AU06], 5.1.1), OSEK/VDX provides ISO/ANSI-C interfaces ([OS04], 2.9). These de-facto[19] standards provide an orientation for programming of a new component like the ETC software and its techniques.

Ada

The introduction of a new component into a high-integrity system will put it under very close scrutiny. Consequently, a potential contribution to acceptance would be to over-achieve the reliability status quo of the domain in question. Direction may be found in domains with use cases of even higher operational safety levels: avionics and defense. There, C and C++ are generally not accepted for mission-critical applications (cmp. [Sto96], p. 227). Instead, e.g. the American Department of Defense commissions much of its work, like the GPS, to be performed in Ada, the language also used for implementing the European Fighter Aircraft systems. Civil users and applications include the European Space Agency, the NASA space station, European Rail switching systems, Boeing 777 and Airbus A320/330/340 ([GH93]).

Ada concepts

Ada, in its current version Ada 2005 ([ISO95], [ISO07]) provides a solid foundation for the fulfillment of the requirements defined. The reasons lie in Ada's view on the following concepts:

Typing – many potential problems may be discovered during compile time due to static and strong typing. Types of all objects have to be declared. They are generally incompatible with each other, so any conversion has to be handled explicitly. Instead of structural equivalence (e.g. size or bit representation), Ada checks for name equivalence when comparing types. Additionally, explicit conversions are restricted to any two numeric types, subtypes of the same type and types derived from the same type. This inherent strictness of the language protects against e.g. assignment bugs from sloppy type conversions so often discovered especially in rushed phases of C/C++ projects.

[19]Or, in the case of AUTOSAR, potentially emerging.

Aspect/Record representation – both approaches I (3.1.3.1) and II (3.1.3.2) include the sharing of resources in a way that components encapsulating sensors, other devices, message busses, services etc. have to interact. To facilitate arbitrary protocol data records of the telematics and automotive domain, the ability to explicitly define the bitwise memory representation of object data is very beneficial for the respective integration. In addition to this convergent[20] encoding, the record representation provides an object serialization fit for persistency purposes.

Concurrency – Ada is a *concurrent programming language*, parallel activity constructs are defined on a language level. In contrast to function/method calls to proprietary, OS specific libraries (like e.g. employed in C/C++), concurrent structures can be expressed directly in Ada syntax. Experience shows us that faults introduced by inconsistent thread behavior, e.g. between two APIs, platforms or operating systems, can be as hard to discover as their effects are unpredictable. The actual semantics of the concurrency constructs are dependent on the implementation of the respective run-time engine (see below), be it provided by the OS or a dedicated, linked RTE. However, parallel programming support provided by the language represents a uniform, binding fundamental for stable processes.

Restrictions and profiles – with the `pragma` compiler directive, especially `Restriction`, Ada supports customized subsetting of the language and tailoring of the run-time system. This way, potentially risky parts of the language can be excluded from the implementation and thusly potential problems avoided, along the way resulting in more efficient, smaller RTEs. For specific purposes, the Ada standard aggregates restrictions together with additional program configuration policies in a run-time `Profile`, defining an alternative mode of operation for a program. The *Ravenscar* profile ([BDV03]) e.g. subsets the Ada tasking model (cmp. [BW98]), aiming at determinism, schedulability and memory-boundedness.

Run-time engines – In the mission-critical embedded environment, besides the program itself, also the run-time engine may be subject to certification. Concurrency features like threads and mutexes are either provided by the operating system, or a dedicated run-time engine – especially if the execution environment does not allow for an OS. For Ada, there are a number of certified or certifiable RTEs available that avoid program constructs which are not verifiable: the *Green Hills Minimal Ada Run-Time* (GMART, [GH081]) conformant to SPARK ([Ba03]; see below), *Real-Time RAVEN* ([Ao06]), *Green Hills Safe-Tasking Ada*

[20] I.e. the ETC software's internal record encoding may converge toward a message format given by another application, e.g. an automotive subsystem.

Run-Time (GSTART, [GH082]) and *Open Ravenscar Real-Time Kernel* (ORK, [Pu01]), the latter three specifically supporting the *Ravenscar* profile language subset.

SPARK, RavenSPARK

In the context of Ada and high-integrity systems we also find *SPARK* ([Ba03]) and *RavenSPARK* ([ST06], [CA08]) for concurrent applications. SPARK is a safe Ada subset, extended by formal specifications, e.g. pre- and postconditions of functions, in the form of annotations. A tool (*SPARK Examiner*) uses these annotations, written as Ada comments to be ignored by the Ada compiler, to check the program for consistency with its specification, well-formedness and to verify the validity of specified properties, e.g. exception-freedom. The focus of this dialect thusly lies on statically proving a program's correctness. We established that, due to the discussed characteristics of the system introduced with this work, it is not possible to completely prove freedom from faults by analyzing the ETC software exclusively. Nevertheless, the SPARK approach provides valid propositions concerning the safety of Ada programs for our context (also see next section).

selecting a reference

Of the languages discussed above, Ada seems to represent the best choice for implementing a high-integrity system. It provides strictness and a host of features that assist in achieving the aims defined for the approach of this work. With other languages evolving toward high-integrity systems, this might change in the future. By selecting Ada for the realization of the architecture introduced here, we present a reference implementation conforming to our requirements defined for an ETC software. This is not meant to restrict alternative implementations in other languages, but merely as a concise description of the system's workings. However, when using another language, the stated characteristics of the implementation should be emulated as explicitly and closely as possible with the respective means available.

3.2.3 Requirements and Programming Restrictions

A number of the Ada concepts mentioned above accommodate our requirements implicitly, on a general level: e.g. strong typing and concurrency language features can be considered beneficial to the reliability of the resulting system. With the ability to subset the language for our purpose, an additional explicit derivation of a corresponding *pragma* set from the requirement definitions of 3.1.4 is indicated.

static implementation

Requirement V calls for a statically determinable implementation. This concerns memory allocation, the determination of variable and function types in assignments and calls, and the binding of methods (dispatching of calls):

Memory allocation – we need to disallow dynamic heap memory allocation; explicit allocation on the application level (with `new`) as well as implicit by any operations realized by the respective Ada run-time environment (e.g. with unbounded types or the creation of task attributes).

Type determination – types of all variables and instances need to be resolved during compile time. Consequently, generic variables to store arbitrary instances of a class hierarchy (*class-wide types* in Ada) are not acceptable.

Method binding – the same applies to dynamic method call dispatching. The code to be called during run-time needs to be determined during compilation.

deterministic behavior

The deterministic run-time behavior required by VI finds expression in the *Ravenscar* profile as mentioned above in 3.2.2. It activates a number of pragmas including restrictions that enforce non-terminating, non-hierarchical tasks with fixed priorities, communicating via protected objects only. Queues are stripped down to one entry per object and single queued up tasks, with simple Boolean entry barriers.

Listing 3.1: Restriction Definitions for the ETC Software

```
pragma Profile(Ravenscar);

pragma Restrictions(No_Allocators);
pragma Restrictions(No_Dispatch);
```

restricting source code

Listing 3.1 defines the restrictions imposed on the architecture implementation. `No_Allocators` takes care of explicit memory allocation, `No_Implicit_Heap_Allocations` covers implicit allocations, but is already included in the Ravenscar profile. `No_Dispatch` supresses dynamic method binding as well as variables of general base class types by effectively excluding class-wide types (`T'Class`) from the program. As an important benefit in an embedded environment, we reduce code overheads, e.g. dispatch look-up tables and record tags.

pointer

We will not completely restrict the use of pointers, or the similar[21] concept of *access types* in Ada. The SPARK approach considers them problematic ([Ba03], 1.4) and this work agrees. However, the HIRTE architecture accepts their employment within the limitations of a static allocation pattern (cmp. [Do03], 6.2) and for an initial linking of statically allocated and declared elements.

3.3 Modeling Approach and Design Notation

The development process necessary to realize a high-integrity system in conformance to the requirements given encompasses a number of topics besides the already mentioned. Regardless of the actual project organization – from

[21]Ada distinguishes *Access* and *Address*, with the former representing a *fat pointer* with additional information like constraints (size, first, last), the latter merely pointing to a memory address.

extreme (cmp. [BA04]) to waterfall (cmp. [Bo81]) – we find the model a crucial development artifact in a multitude of roles. Thus, we have to establish a formal approach to formulate the system in order to pursue maximum model useability: as specification, origin of the implementation, basis for validation and a structured means to communicate and elaborate on the software.

3.3.1 Discussing the Software Model and its Purpose

modeling and the UML

First, on modeling in general. While some time ago a heated discussion about universal approaches versus domain-specific modeling languages broke out, we will only consider the UML ([OMG091] and [OMG092]) here – it is customizable and a clear preference of all our industrial customers. Many of the published profiles, prominently the UML Profile for MARTE ([OMG09], Modeling and Analysis of Real-Time Embedded Systems; previously the schedulability, performance, and time specification [OMG05]), cover an impressive width of real-time system aspects, providing numerous diagrams, stereotypes and tagged values.

model elements and the need for semantics

When designing an actual product however, one has to keep in mind that every single element used in the respective model has to transport an unambiguous meaning. These semantics cannot be (exhaustively) defined outside of the corresponding development itself, as especially the operational semantics are dependent on an actual, if possibly abstract, machine. Consequently, some run-time level semantics definition has to be part of the development preparation.

selecting sensible syntax elements

So – as the above preparations consume time and money –, an initial thorough selection of language units (cmp. [OMG092], 2.1) and their syntax elements is indicated: what is really necessary to describe the system, and what purpose does each element serve during the development process? Take, for example, time constraints in statechart diagrams (cmp. [Do04], chap. 2.6). Once the detailed time behavior of the application's processes has been defined (again, it takes a lot of work), what will become of the respective model attributes? Will they just be a part of the documentation someone may refer to later if something went wrong to find out why? If they are a requirement specification, how will the programmer be able to ensure a corresponding implementation? In this context it is important to note that a telematics software might be located on a layer too remote from the hardware to actually control or even directly adjust to the hardware run-time behavior. Here, time constraints are useful for validation purposes as well as monitoring during operations.

role of a model

[Te02] made an interesting point concerning the UML in high integrity systems, stating that in order to be useful, it would have to be effectively isomorphic with the target programming language. This is understandable, but we would not go that far. Instead, again we ask for the purpose of the model. A direct, explicit semantical correspondence could result in "programming in UML" – quite impractical, as any experienced software engineer would probably reply. The contribution of a model and its language to a project can well be simple, but still valuable. The model may just represent a structured documentation; a class diagram a kind of map for the developers over the software lifecycle. Behavior diagrams may be produced for critical processes only, so that a general understanding is granted for all stakeholders. Additionally, with the selective definition of domain-/project-specific semantics, model transformation techniques can significantly add to completing the development process. Without the need to express everything in the model or the UML, consider e.g. a statechart to be checked for consistency or be transformed into code. Predefinitions and parameters of its run-time environment and structure might be described elsewhere, e.g. encoded in a transformation tool and associated mapping rules.

experiences

In industrial telematics projects we use the UML very successfully in the sketched way: selective employment of diagrams in combination with selective but exhaustive domain-specific semantics definition, transformation rules for pragmatic model checking and code generation.

3.3.2 Syntax – Views, Diagrams and their Elements

Like the requirements reflect in implementation characteristics (see 3.2), they influence the software model in a number of ways. Obviously, we have to specify both the static view, or structure ([OMG092], part I), and dynamic view, or intended behavior ([OMG092], part II), of the ETC software architecture. By selecting the UML, a work is confronted with a multitude of potentially useful syntax elements, i.e. means to describe the system for the various stakeholders.

applied diagrams

To express the fulfillment of the defined requirements concerning the model, we use elements of *class*, *component* and *statechart* diagrams[22]. Their adoption is further differentiated for closeness to the implementation, i.e. whether the structure represented by the diagram merely defines a pattern, or is meant as input for model transformation. The former case is subject to interpretation by a software architect and thusly still allows for a degree of freedom, the latter requires unambiguous mapping rules to support automation.

requirements impacting the model

Impact of the requirements on syntax element selection is marginal but worth mentioning. Besides the straightforward use of class, component and statechart diagrams to illustrate fulfillment of II and IV, requirement V has more subtle consequences. To ensure a valid translation of diagram structures into statically determinable code, i.e. its records and classes, we have to restrict associations to generic, base classes. Indiscriminate application would potentially result in the need for dispatching in the corresponding software.

[22]Various other design and implementation level diagrams can be found during the course of this document, e.g. sequences and deployments. They are used to illustrate specific topics or aspects, but are no immediate representation of the core result – the architecture and its implementation – of this work.

Note that it is unnecessary to disallow class inheritance in general; a selective approach is sufficient.

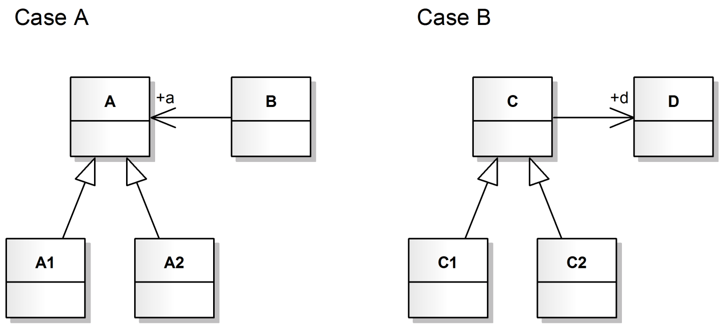


Figure 3.18: Class Structures influencing Dispatching

model implications for dispatching

In fig. 3.18, case A illustrates a problematic constellation. On invoking an operation of *a*, the RTE needs to determine the type of object bound to that attribute (*A1*, *A2* or *A* if it is not abstract), check if the operation is defined for that type and then proceed to the specific implementation for execution. As long as the inheriting classes do not overload *d*, case B is acceptable, as it does not imply dispatching. A syntax restriction suppressing dispatching could thusly be enforced by checking the model for corresponding syntactical patterns as given by case A.

3.3.3 Semantics

Conforming to [OMG092], chapt. 6.2 we need to define run-time semantics for the selected syntax elements. These semantics definitions have to refer to a specific RTE and support fulfillment of the introduced requirements, i.e. the resulting architecture description can exhaustively describe an implementable solution. Here, we take a rather pragmatic view on the term of semantics, generally denoting the meaning of specific elements during run-time, facilitating a realization in a programming language.

UML semantics in Ada

Relating structural and behavioral syntax elements of UML to our selected language Ada implies the explicit mapping to run-time representations and operational processes. Due to the interdependencies between model and implementation induced by the restrictions (as discussed in 3.2), applicability of both static and dynamic language structures is limited for this work; the Ada language is narrowed down to a subset.

mapping of aggregations

In consequence, instead of generically mapping basic UML, e.g. classes and primitive types of the Core library [OMG091] part II to Ada tagged records, tasks and other types ([ISO95]), we describe transformations of aggregations and complete patterns. This way, by considering the context of the actual intended application domain, the impact of safety constraints on architecture and implementation should become clearer. Also, generic transformations of UML to Ada have been handled before; e.g. [KK01] discusses and defines a mapping for reactive, distributed systems, but without regard to our high integrity considerations.

statechart mappings

A special case is given by statecharts. Already, numerous solutions are available for operational semantics and conforming statechart model transformations into process algebras or programming languages. [Sa02], describing yet another approach, compared a representative set of standard state machine implementations besides its own *Quantum* framework: *nested switch statement*, *state table*, *State design pattern* ([GHJV95], p. 305) and a combination of the previous called *optimal FSM implementation.* An Ada state machine is presented in [Sa00], which is primarily a plain variant of nested switches.

specific requirements for state machines

Requirements II and IV established state-based components for our architecture. Like stated in the previous section, the use of statechart diagrams in the software model can merely illustrate conformance to the requirement. Actual fulfillment is achieved by the realization of components as state machines. At this point, requirements V and VI take effect. They introduce state machine implementation characteristics relevant specifically for this work: the machine program structure is free from dynamic memory allocation and dynamic binding. Overhead event processing, interleaving or concurrency mechanisms support (or at least do not hinder) static priority patterns (cmp. [Do03], 5.9, 7.5). Additionally, for all components implementing services to meet the stipulation of conformance to a state machine organization, we imply a corresponding expressiveness of its structure. As the application features complex business logics (cmp. 2.3.8), construction of the state machine includes hierarchy and (de-)composition, i.e. structural means to cope with complexity, similar to a regular programming language. Related to the common approaches this means:

- The nested switch and state table approaches are too restricted in their expressiveness. With just simple states it would be hard to formulate complex services. Introducing composite states and direct transitions leads to hardly manageable and suboptimal program structures, e.g. hard-coded transition and action chains over state hierarchies. Still it should be noted that the switch structure is a very efficient implementation method for flat lexical scanners and parsers (as we will see later in our reference implementations).

- Both the State pattern and the Quantum framework heavily rely on dynamic OO features. Examples: The State pattern subclasses a generic *State* class, overloading *handle()* methods resulting in regular dispatching. Events in the Quantum framework are dynamically allocated, passed and propagated bound to generic variables (cmp. [Sa02], 8.5). Note that these dynamics are inherent to both approaches, and not adaptable to a static architecture without substantially changing the concepts.

- While set in the context of embedded systems, the Quantum framework aims at providing a universal solution. As such, it is burdened with the comprehensive realization of any UML statechart elements. This collides with our intended selection as motivated in the previous sections. In over ten years of designing ETC and VAS software, no use case realization ever depended on e.g. *deferred events*, *history states* or *behavioral inheritance*. They require a significant management overhead, in consequence introducing risks due to high complexity, but in turn provide little gain for our domain.

As a result, this work tailors a specific approach in conformance with the ETC domain-specific requirements, in the case of a state machine namely static determinability of memory consumption and function calls.

3.4 Chapter Conclusion

The previous chapter affording us substance to analyze, the generic requirements of reliability and economy could be primed for the design of a corresponding architecture.

hazard classification

To detail the meaning of reliability in the context of the ETC domain, we asked for reasons that could disrupt regular and correct operations of the system. Application of a fault tree analysis – with a root node of *faulty toll atom processing* – to the introduced ETC structures and processes identified a set of these hazards. They originate from a variety of sources, but corresponding to their symptoms can be generalized and classified conforming to

the ETC aspect of the application – *locating*, *charging*, *active data storage*, *communications*, *enforcement* and *security* (as introduced in chap. 2.3),

the system area or level – *organization and business processes*, *quality of service*, *data* and *implementation* and

the system element – *the system itself*, its *interfaces* and *actors* interacting with it.

reliability requirements

Based on these generalized correlations, relations to the system deployment and classifications of effects on the ETC software, we deduced a set of hazard-induced requirements for the architecture. In the process, it was an important insight having to accept the general concept of *a safe application in an unsafe environment.* In consequence, the application itself is neither in the position of being able to fully control its environment, especially beyond the OBE, ensuring overall ETC system stability. Nor is it sufficient for a complete tempering, i.e. operational validation, of the whole system to verify the ETC OBE software. To answer and compensate for this, the software architecture needs

- a central element controling or at least monitoring all application processes (*requirement I*),
- to pattern all processes as state automata to explicitly manifest uniform behavior (*requirement II*) and
- to continuously log a (real-)timely state configuration of this behavior (*requirement III*).

Adhering to these requirements enables the validated application to prove its correct operation even if interaction partners in its environment exhibit faulty behavior (the OBE hardware, center components, communications interfaces etc.).

achieving cost-efficiency

After the question of reliability, we related economy to the ETC software domain, based on the optimized utilization of the necessary infrastructure. To avoid the need for dedicated – and thus expensive – solutions, three cost-driven approaches considered running the ETC application in non-exclusive RTEs:

Open ETC infrastructure describes value added services (VAS) on ETC platforms.

Automotive integration uses ECU networks of modern vehicles as RTE for ETC modules.

Interoperability suggests using multi-operator software, multi-application OBE or application switching to participate in more than one ETC service, respectively communicate with more than one ETC operator. In this context, we detailed an original smart card-based scenario to further substantiate the economic view.

economy requirements

Generalizing the requirements of these approaches led to the question for stable resource sharing. Their realization has a number of preconditions. Consequently, the economy-driven architecture in the ETC domain needs

- to organize all services in the form of components – encapsuling the behavior, interacting over message queues (*requirement IV*),
- to be implementable as completely static structures and operations (*requirement V*), avoiding dynamic memory management, dynamic typing and dynamic binding,

- to allow for deterministic concurrency and scheduling (*requirement VI*), i.e. it should not be dependent on schemes considered unsafe, like dynamic priority patterns (cmp. [Do03], 5.10) or priority inheritance patterns (cmp. [Do03], 7.13), and
- an architecture-fixed feature to safeguard the persistency of the application's state configuration at any time (*requirement VII*).

An ETC software conforming to these requirements reduces the risks of side effects to concurrently present processes and itself will not (or at least is sufficiently unlikely to) impede sharing run-time ressources.

model versus program

The seven requirements substantiating reliability and economy for the ETC software architecture set a scope for its design. However, requirements V and VI immediately affect the implementation. Thus, we established interdependencies between model and program that reflect each view, i.e. the model design may not arbitrarily disregard the implementation level, as this might lead to dead ends – unimplementable structures – later. As the work has to provide reference implementations of crucial parts, the chapter motivated the selection of a suitable language – in this case Ada.

modeling and programming rules

We then proceeded to concretize the requirements V and VI as programming restrictions for the Ada sources and RTE, resulting in three *pragma* directives. Correspondingly, we described modeling restrictions and guidelines for the selected UML syntax elements and their semantics conforming to the defined requirements. Regarding requirement II (*system state automata*), reasons for a genuine approach to the implementation of state machines were given, discarding existing solutions in the process.

Implicit to these measures is a general *correctness by construction* approach (similar to [Ba03]), as we strive to support program correctness by proper techniques employed from early on in the development process. Experience shows us that by restricting dynamic memory management[23] alone, errors are reduced significantly, especially in early builds – the program construction effectively excludes problems like memory leaks.

With this chapter, an ETC software architecture – generalizable to mission-critical telematics systems – finds concrete requirements to further and attain robustness and cost-efficiency in a corresponding application. Matching rules for modeling and implementing the domain-specific architecture complement the seven requirements. Together with the structures and processes of chapter 2, the next chapter can now lay the foundation for the architecture itself.

[23]Experience also tells us that this – by all means – includes garbage collection!

Four

Fundamental HIRTE Patterns

... in which we provide the schematics and building blocks of a reliable and economic mission-critical telematics software. The bottom-up construction starts with a number of elemental stereotypes for differentiation from common realizations and preparatory conceptual advancements. To emphasize reusability and orderly decomposition, separate patterns are given for specific aspects of the architecture, e.g. static state machines and logging. Each pattern also describes its Ada reference implementation.

The aspects and business processes of chapter 2, and the decisions and requirements of chapter 3 regarding the architectural design, provide us with sufficient information to specifiy a solution for an ETC OBE software. However, it seems rewarding to dwell on the transition – represented by this chapter – from requirement (respectively design decision) to realization. Instead of directly describing a specialized application, we first elaborate on the underlying patterns and mechanisms. These are qualified to provide a basis for a reliable and economic ETC software, but also for similar distributed architectures of related telematics domains, e.g. healthcare or smart energy grids.

This corresponds to the notion of HIRTE: A High Integrity Run-Time *Environment* may generally compose a set of components, each representing a mission-critical operational aspect. While any composition needs to conform to the established high integrity requirements, the architectural concept needs to allow for variants. As was established in 3.1.1.10, an ETC application has little influence on its hardware and OS platform. Consequently, HIRTE, in the given case directly implementing the ETC functionality, has to adapt to fundamental underlying RTE parameters like available memory, multitasking (or lack of) – while still maintaining the invariant HI conformity.

4.1 Structural Elements

The UML *stereotype* mechanism (cmp. [OMG092], 18.3.8) allows the extension of metaclasses. Thus, the stereotypes given here represent a set of basic elements that serve two purposes: refinement and differentiation.

By extending a metaclass, we gain a specialization for our domain. Adding specific attributes refines a metaclass to better fit into our architectural concept, stipulating information required by the application. Furthermore, the stereotype instances are easily identified by both a human software engineer and a model transformation tool. I.e. instead of checking for various characteristics determining a domain-specific type, one can directly refer to a stereotype regarding its supposed implementation scheme.

4.1.1 Behavioral Stereotypes

The common view on object-oriented software sets procedures and functions in the context of classes, i.e. they represent methods defined to operate on abstract data structures. Aside from their actual definition, it is possible to declare class attributes that point to methods of other classes.

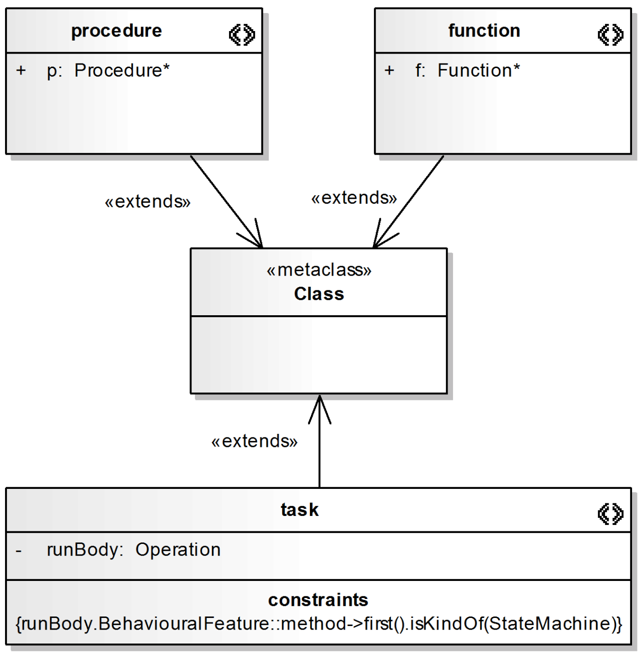


Figure 4.1: Stereotypes representing Tasks, Function and Procedure References

function and procedure

The stereotypes in fig. 4.1 additionally allow class attributes defined by associations to classes that represent *function* and *procedure* (function without a return value) references. The name of an instance of these stereotypes is determined by the name of the function or procedure, respectively. Its signature may be specified by the tagged value. Whether or not the programming language of the implementation restricts functions and procedures to class operations, and if that operation would be defined directly in the extended class or elsewhere, is not considered relevant for application. In such a case, the tagged value type would be *Operation**.

task

The *task*[1] stereotype is introduced to facilitate identification and language-specific conversion by model transformation tools. It designates an active class, i.e. it represents an execution unit during run-time. Therefore, it requires an *Operation* to run its body. This body shall be specified as a state machine, associated per definitionem with the first element of the *method* set given by the meta base class *BehavioralFeature*. Whether the task implements a thread or process, its details of activation, execution and finalization, depend on an actual RTE.

4.1.2 Data Structure Stereotypes

As discussed and established in chapter 3.2, a general abstraction from an implementation is desireable, but not always advisable. Thus, in addition to the induced restrictions, the implementation requires complementary information from the model.

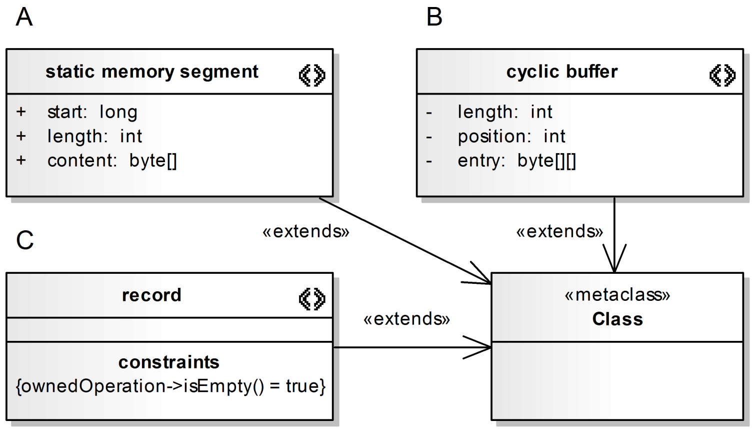


Figure 4.2: Stereotypes representing Data Structures

static memory segment

The *static memory segment* stereotype as given by fig. 4.2 A enables the model to specify direct memory usage, instead of a class or other variable, that is bound by a compiler. A corresponding instance represents a fixed memory segment with the tagged values

[1]In this case akin to the *task* type as introduced by Ada ([ISO95], section 9).

start – the starting address as an assumed *long* value,

length – the size of the segment in bytes and

content – an array of bytes with its binary content.

All attributes are deliberately declared public to emphasize the immediate access character of the instance. No detours over *get* and *set* methods are intended. The option to explicitly specify an actual starting address (at least symbolically) in the model is especially useful for our domain, as it allows referencing a static storage for crucial content.

cyclic buffer

The *cyclic buffer* stereotype (fig. 4.2 B) represents a memory segment that is organized as a queue, or FIFO, for *length* binary strings, overwriting the earliest *entry* after reaching the condition *position = length.* We introduce this stereotype, because the cyclic binary buffer is a recurring entity in ETC domain models, which has to be implemented in specific ways depending on the underlying platform, e.g. with regard to dedicated memory segments (see above).

record

A stereotype for a *Record* structure allows the declaration of plain variable tuples, excluding operations by applying a corresponding constraint to the extension of the metaclass.

4.1.3 Distributable State Machine Fragment Stereotype

UML state machine restrictions

The UML state machine metamodel defines (besides a number of *pseudostates*) *simple states*, *composite states* and *submachine states* (cmp. [OMG092] 15.3.11), with the submachine being semantically equivalent to a composite state. Any actions associated with states or transitions aggregated by (regions of) the machine may access and operate on attributes defined by a *classifier context*, an association with a *behaviored classifier*, or the association *specification* with a *behavioral feature* that owns the state machine (cmp. [OMG092] 15.3.12 and 13.3.2), respectively of which the machine specifies the *method.* This implies that any state taken out of context of its machine has to be considered incomplete, as references to required features become invalid.

introducing the self-contained state

In order to concisely extend this model for our purposes, we need a formal definition. Similar to our description given in [Ste032], a state machine $SM = (S, substates, regions, E, P, E_p, A, actions, V, G, T)$ consists of

- a non-empty, finite set of states $S = S_s \cup S_c \cup S_{cc} \cup S_{cc,r} \cup S_{sc}$ with simple states S_s, composite (OR) states S_c, concurrent composite (AND) states S_{cc} and regions $S_{cc,r}$ of AND states. Additionally, we introduce the *self-contained state* S_{sc}, which is defined in detail below.

- Functions

$substates : S_c \cup S_{cc,r} \cup S_{sc} \to \mathcal{P}(S_s \cup S_c \cup S_{cc})$ (mapping of a set of aggregated states to a composite state or region of a concurrent composite state; with $\mathcal{P}$ the power set) and

$regions : S_{cc} \to \mathcal{P}(S_{cc,r})$ (mapping of a set of regions to a concurrent composite state).

- A finite set of events E, parameters P and a parametrization relation $E_p \subseteq E \times P \times ... \times P$ (parameterless events shall be denoted simple events E_s).
- A finite set of actions $A = A_{event} \cup A_{var} \cup A_{gen}$ with triggers A_{event} of elements of E, value assignments A_{val} to elements of V and generic actions A_{gen}, e.g. invocations of methods, procedures and functions defined outside of the scope of the state machine, and a relation $actions \subseteq S \times A^n, n \in \mathbb{N}$ that specifies the sequence of actions to execute in a state.
- A finite set V of variables accessible by the state machine.
- A finite set of guard conditions G that apply relational operators to variables of V and values.
- A relation $T \subseteq S \times E \times A^n \times G \times S, n \in \mathbb{N}$, defining a transition from one state of S to the same or another $s \in S$ on an event of E and/or a true condition of G, triggering a sequence of actions in A.
- A self-contained state is defined as tuple $S_{sc} = (S_\sigma, substates_\sigma, regions_\sigma, E_\sigma, P_\sigma, E_{p,\sigma}, A_\sigma, actions_\sigma, V_\sigma, G_\sigma, T_\sigma)$, with

  $S_\sigma \subseteq S, substates_\sigma \subseteq substates, regions_\sigma \subseteq regions$ the substates contained in S_{sc},

  $E_\sigma \subseteq E, P_\sigma \subseteq P, E_{p,\sigma} \subseteq E_p$ the (parametrized) events triggered by elements of $A_{event,\sigma}$ or triggering transitions in T_σ,

  $A_\sigma, actions_\sigma$ the actions executed by states in S_σ,

  V_σ variables evaluated by G_σ and accessed by $A_{var,\sigma}$ and $A_{gen,\sigma}$,

  G_σ guard conditions evaluated by T_σ and

  T_σ transitions $(s_{source}, e, a, g, s_{target})$ with $s_{source} \in S_\sigma$ and $s_{target} \in S$ (targets may lie outside of the self-contained state).

Note that the above definition focuses on the algebraic structure of a machine. Specifics of run-time semantics – invoking actions (on entry, exit), event handling, transition firing, updating variables (especially those referenced by a self-contained state) – are not yet significant, but will be treated in detail later.

valid state machine fragments

The concept of a self-contained state establishes a fragment that is valid and meaningful without its ambient state machine. Validity in this context pursues closed interpretability, i.e. the fragment contains sufficient elements and associations between them to result in a meaningful structure when a semantics definition is applied. Finally this closedness leads to isolated executability: binding the fragment specification to the run-time semantics of an actual RTE enables a transformation from a defined start state to some final state.

utilizing the self-contained state in UML

To harness this concept in our HIRTE architecture, the stereotype *distributable state machine fragment* migrates the self-contained state into the UML model. In the process, the notion of self-containment serves as foundation for distributability: the isolated state machine element can be transferred to another RTE. Here, this fragment has to provide an external component with sufficient information to execute the self-contained state and return it on reaching a transition beyond its scope, i.e. leaving its vertex, or some other defined condition (e.g. timeouts, errors, unresolvable references).

The stereotype as defined in fig. 4.3 extends the metaclass *State* and adds the following tagged values.

Property array *sm variable subset declaration* – defines a subset of the attributes given by the state machine's owning structural element. Each array element of a corresponding fragment instance specifies name, type and a value, thus representing a copy of the original variable. Instead of directly mapping the full property set, the tagged value allows the explicit *declaration* of a limited set of variables. This way, the array can constrain itself to the attributes actually accessed by the state machine, reducing memory consumption of a fragment object. Furthermore, while it is still possible to generate the set from a model query[2] of the associated class, as the stereotype's concept implies handing data over to external systems, being able to exclude variables from the set provides means to restrict accidental distribution of critical values (e.g. keys or certificates).

Action array *sm action requirements* – specifies the actions required in the execution of the fragment. These actions may be derived from an iteration over the the *entry*, *exit* and *do* associations of each state owned by the machine, where the type of the *Behavior* evaluates to *Action*. In contrast to the variable array, the requirements merely represent references to actions, not their implementation. Consequently, the action descriptions may serve to check in advance if a RTE is able to execute a fragment, i.e. if it provides realizations of the required actions. Congruously, confining specific actions to a fragment when modeling the state machine assures external execution, relieving the local RTE from having to implement them.

[2]The query sketches in fig. 4.3 are not intended to be compilable OCL terms, but specify the model elements to iterate and conditions for identification of the desired attributes. They are kept abbreviated for better comprehension.

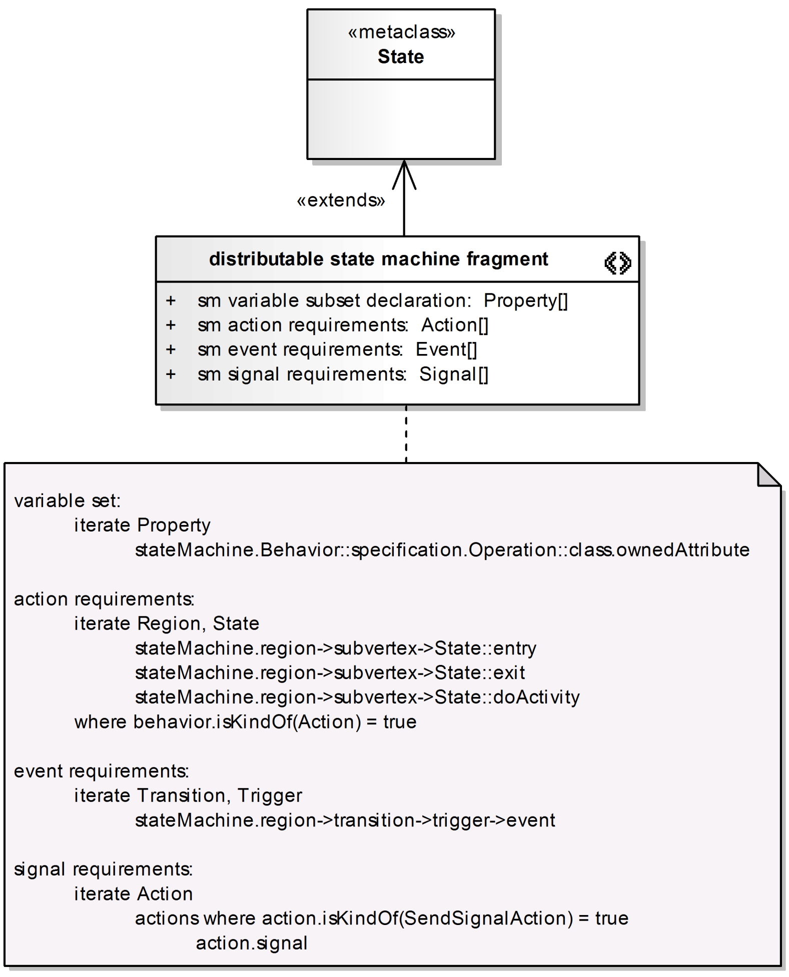

«metaclass»
State
«extends»
distributable state machine fragment
+ sm variable subset declaration: Property[]
+ sm action requirements: Action[]
+ sm event requirements: Event[]
+ sm signal requirements: Signal[]
variable set:
iterate Property
stateMachine.Behavior::specification.Operation::class.ownedAttribute
action requirements:
iterate Region, State
stateMachine.region->subvertex->State::entry
stateMachine.region->subvertex->State::exit
stateMachine.region->subvertex->State::doActivity
where behavior.isKindOf(Action) = true
event requirements:
iterate Transition, Trigger
stateMachine.region->transition->trigger->event
signal requirements:
iterate Action
actions where action.isKindOf(SendSignalAction) = true
action.signal

Figure 4.3: Machine Fragment Stereotype with Model Queries

Event array *sm event requirements* – specifies the events expected in transitions owned by the fragment. In a similar fashion to the actions, queries of the state machine model yield the events associated with triggers of transitions. Here also, it is possible to reference events that will be generated exclusively by the external RTE.

Signal array *sm signal requirements* – refers to signals that may be generated by a *SendSignalAction* subset of the above actions. As long as these actions are available on external execution, the fragment will be able to process its states. However, there is a potential indirect effect concerning signal reception by the external RTE and its reactions. Interactions between machine fragment and environment depend on signal acceptance and target presence, i.e. the signal array combined with the above event array describes the prerequisites for consistent information interchange.

Section 4.2.4 (the distributable state machine fragment pattern) will elaborate in detail on the run-time characteristics and handling of machine fragments.

We suggest the notation as given by fig. 4.4 for distributable states in a statechart diagram. A corresponding symbol complements and distinguishes the state rectangle. Four compartments contain the tagged value arrays of the embedded variables and the expected action, event and signal realizations. Depending on the statechart's layout and the quantity of compartment entries, alternatively, the tagged values may be specified as notes attached to the state (cmp. [OMG092], fig. 18.19).

4.2 Elementary Patterns and Structures

We use the principle of *design patterns* (cmp. [GHJV95]) to provide further, more complex building blocks of the HIRTE architecture. An emphasis lies on mechanisms that enable efficient, HI-preserving conformance to requirement II (3.1.2) – the implementation of HIRTE services as state machines.

All presented Ada sources compile with the *Ravenscar* profile and additional restrictions in force as defined in listing 3.1. The implementations need to refrain from object-oriented reusability structures and concepts that might compromise static determinability. Instead, the given systematic composition of the sources implies (at least partial) generation from a model. This way, even without certain comforts of OO, the efforts of feature changes are alleviated.

4.2.1 Statically Resolvable State Machine Pattern

4.2.1.1 Abstract

relation to UML statecharts

In 3.3.3 we discussed and motivated the need for a state machine implementation tailored to the HIRTE requirements. The *Statically Resolvable State Machine* provides a realization blueprint for the behavior of an active[3] component. Construction of the pattern aimed at a lightweight, safe and manageable structure. A machine instance may be based on a UML statechart description for manual or automatic transformation, i.e. the pattern recognizes a subset of UML elements:

- *simple states* with *entry* and *exit actions*,
- single region *composite states* with *entry* and *exit actions*,

[3]This is the focus here; nothing should prevent the pattern to apply to passive modules also.

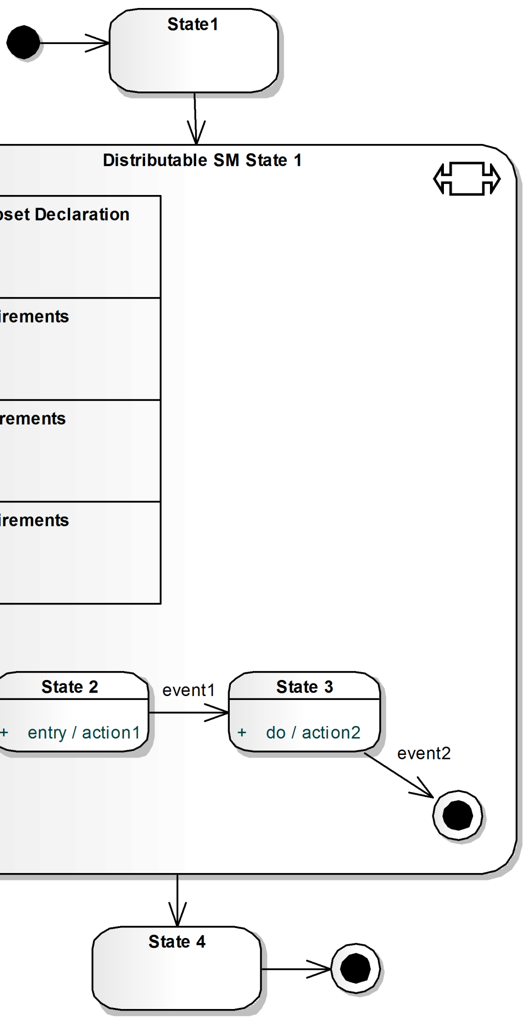


Figure 4.4: Distributable State Machine Fragment Notation

- *initial* and *final pseudostates*,
- direct, high-level, explicit and completion *transitions* with *actions*,
- triggered by *events* resp. message reception and/or *guards*.

achieving light weight

While the selection may appear reduced compared to other works, it reflects a conscious decision that balances a mandatory overhead to process the state machine structures against practical gain (also cmp. section 3.3.3). The most disputable (in the sense of fundamental) ommission might well be orthogonal regions, i.e. concurrent composite states. However, we come to a similar conclusion as [Sa02] (cmp. *object composition* in 5.4.3 there): multiple regions require a heavyweight management mechanism and can in our case be systematically substituted by multiple tasks or additional machine structures in the main loop (shown below, see implementation). This is also confirmed by industrial experience with complex state machine models in the treated domain that yielded not a single automaton dependent[4] on orthogonal regions. Consequently, the pattern abandons them to keep the architecture lean.

supporting statical resolvability

A safe realization is aided by statical resolvability. Implied is a general adherence to the restrictions established in 3.2.3. More precisely, an implementation conforming to the pattern allows determination of memory usage, variable and procedure binding at compile-time. It avoids dependencies on dynamic mechanisms in its structure, proven by the actual source code given in this section. There is a distinction between *checks by the compiler* and *checks at compile-time* that are not necessarily performed by the compiler itself. This is a concession to the tentative permission of pointers, colliding with other high-integrity approaches like SPARK ([Ba03]), which excludes them from its Ada language subset. But pointers simplify our reference Ada sources considerably, hence they help to convey the principles of the pattern workings. By sacrificing a measure of decomposition, it is feasible to eliminate pointers in a pattern realization, which is sketched at the end of the section.

dealing with pointers

In the coded solution given here, all pointers refer to statically allocated objects. Pointers defining the state machine structure are only assigned once on initialization. The associations between states, transitions, guards and actions stay invariant once established and can be traversed deterministically by an interpreter procedure. Still, formally the program handles these pointers at run-time. As a result, the compiler has no approach to check these constructions for correctness. Even so, it is possible to assert *completeness* (all state machine elements required for execution/interpretation accounted for), *coherency* (or *well-formedness*, associations between elements valid) and – in the case where some model is available for reference – *consistency* (no deviations between specification and implementation). If the initialization is contained obligatorily in a designated procedure (cmp. reference sources below), this may be achieved dependably by checking the corresponding source by proprietary means.

pattern model and code

Correspondences between UML statecharts and the elements of our state machine pattern facilitate source code generation based on a model. A model-driven development process (cmp. e.g. [BCK03]) benefits from such a feature, as it may accelerate specification/modeling and programming cycles. It offers an option to delegate management of the state machine structures to a qualified tool ([We07] for an overview). Nonetheless, industrial telematics domain experience tells us that an actual complete round-trip software engineering – UML modeling, executable code generation, code modification (test cycles

[4]Actually, quite to the contrary: specifications became unwieldy, their purpose and implementation ambiguous.

omitted), model updating – is in most cases considered too costly to set up (cmp. 3.3.1). In consequence, expressing the pattern structures in a programming language needs to result in source code that can be authored, understood and maintained without the aid of modeling or other advanced software engineering tools.

manageable results

The concept of a state machine, illustrated as a statechart, is a proven approach to cope with the complexities of component behavior: a vehicle for specification, discussion and implementation. When mapping a state machine metamodel to a state machine pattern, one is (simplified) confronted with the trade-off between preserving the original structures and accepting heavy processing overhead, or altering the structures and optimizing them for more efficient processing. The *Statically Resolvable State Machine Pattern* strikes a balance by selectively introducing auxiliary constructs – state reference chains – while keeping a basic equivalence – states, transitions, actions, guards – intact for facilitated recognition.

4.2.1.2 Structure

The structure of the pattern is described by fig. 4.5.

4.2.1.3 Collaboration

- *HIRTE Component*
  An active *task*, implementing a service component which needs to conform to high-integrity requirements and whose behavior may be expressed as a state machine. During execution, it can read and write variables declared in its *Attributes.* For interaction with other components, the task accesses an *Input* and *Output Queue.*

  The component stores the IDs of its complete state configuration in the array *Active States* with regard to hierarchy – more than one state can be active. Unique state IDs are defined by the enumeration *State ID. Current Depth* serves as indicator for the hierarchy level and is thus equivalent to the write index of the *Active State* array. The currently processed state is referenced in *Current State*, the currently fired transition in *Current Transition.* Per definitionem, i.e. as a precondition, the first state to process is denominated *Initial. Current Chain* references a state in a sequence of composites if a (high-level or explicit) transition traverses a state hierarchy.

  Coming from the *task* stereotype's *runBody* operation, a *Run Machine* operation constitutes an iteration, e.g. an outer loop, over the state machine elements. The *Check Transition* operation implements a mechanism to determine the firing of a transition, i.e. evaluation of guard conditions and defined events.

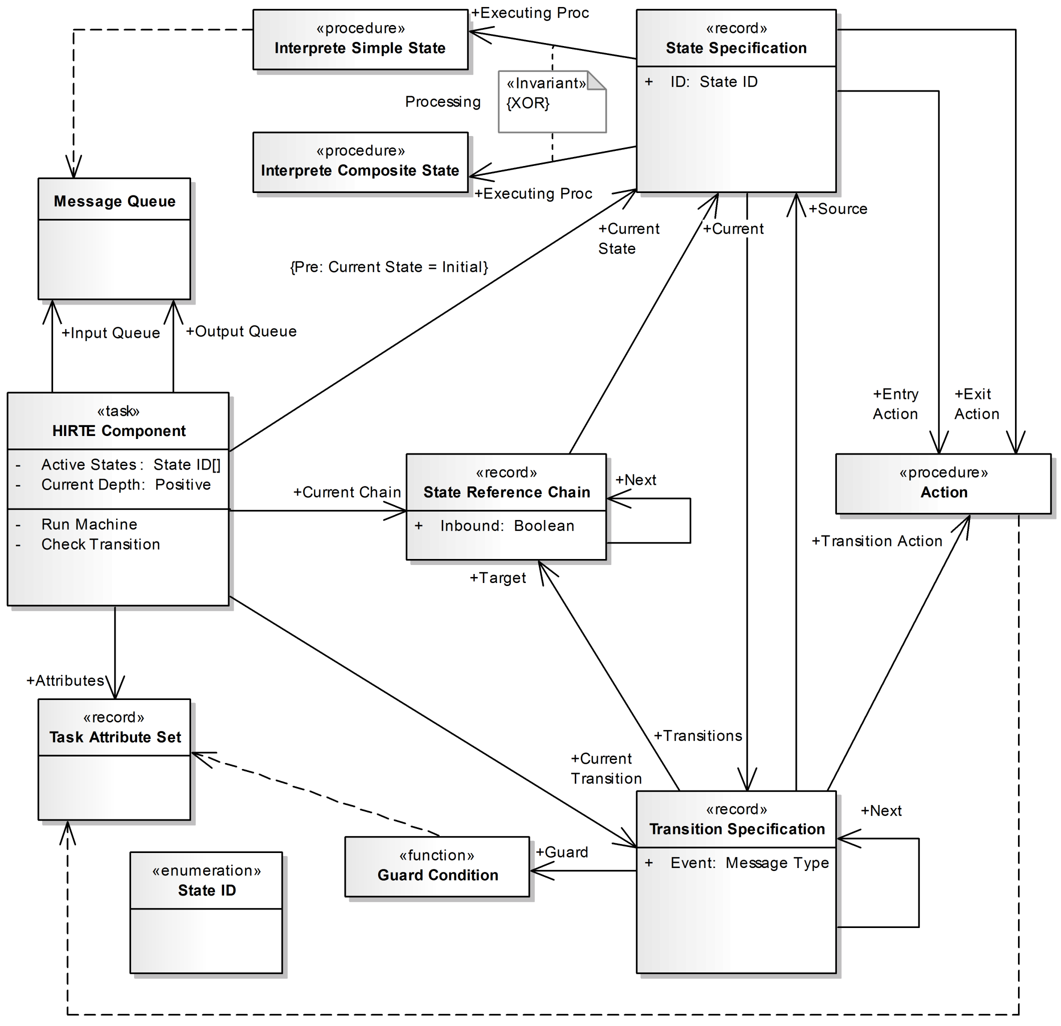


Figure 4.5: Statically Resolvable State Machine Pattern Structure

- *Task Attribute Set*
  A *record* declaring and composing all variables – standard, array or structure types – required in the execution of the service the component realizes. This directly implies a restriction for any procedure implementing activities and data flows of the component: it may only refer, i.e. read from and write to, this global set local to the task.

  Introducing the set to the pattern goes a long way toward requirements V and VII: it avoids implicit or local allocations and, when properly applied, can completely contain variables relevant to the service runtime state as contribution to a persistent system state.

- *Message Queue*
  Provides means for interaction between components, conforming to some organization principle and parameters. While the *Ravenscar* restrictions refer to Ada RTE entities (e.g. tasks and their internal queues), their policies should reflect in the architecture itself: besides size, internal structure and mutual exclusion, the application-level queue also needs to refrain from dynamic priorities (cmp. [BW98], 12.1.4).

  The actual messages and formats are implementation-specific. On the component implementation level, especially for guard conditions, we assume the following relationships between *messages*, *events*, *calls* and *signals* (cmp. [OMG092], 13). An event is universally a *message event* ([OMG092], 13.3.18), i.e. a message of specific type is read[5] from the queue. A signal is represented by a message, both for sending and receiving. Same applies to a call; if a service provides methods, they would be invoked by messages, with their return values also encoded as a reply message.

  This implies that the message queue is the *exclusive* interface to interact with a component in the HIRTE[6]. Only elements of the HIRTE itself (e.g. the state tracer introduced below) may override this mechanism.

- *State Specification*
  Defines a simple or composite state of the machine, identified by the *ID* of the *State ID* enumeration type, or the pseudostates *Initial* and *Final* with the corresponding IDs. If specified, *Entry* and *Exit Action* reference procedures implementing the actions to execute on entering and leaving the state.

  Transitions references a linked list of all outgoing transitions, direct and indirect, i.e. high-level origins. The order may express hierarchical precedence, depending on the implementation (see next section for an example).

  Executing Proc discriminates between state types by assigning a procedure to process the specificied state either as simple or composite. Resolving the association of an attribute with two subtypes could technically be classified as dispatching. Note however, that HIRTE handles this with an explicit assignment and subsequent call, no involvement of an interpreter procedure base class and consequently no dispatching overhead required in the RTE. Therefore, while avoiding the constellation of 3.3.2 case A, the pattern achieves a similar result, as demonstrated in the reference source 4.1.

[5] In contrast to the mere entry of the message in the queue, which is not immediately detected by the component state machine itself.

[6] It is still possible to induce unchecked input from outside the HIRTE sphere of influence as was established in 3.1.1.10; see *Actions* below.

- *Transition Specification*
  Establishes a complete transition between two states, originating from *Source*, with the *Target* designating a simple state or a partial transition to a composite state, if the transition enters or leaves one or more hierarchically arranged borders. *Event* specifies the type of message that could trigger the transition on reception, i.e. reading from the component's in queue, if the boolean function referenced by *Guard* evaluates to *true*. For a firing transition, the procedure referenced by *Transition Action* is called.

  The specification represents a list of outgoing transitions, so *Next* references the successor.

- *State Reference Chain*
  Indirectly[7] establishes a hierarchical order between states by separating a transition into successive steps *Next* into states referenced by *Current*. In contrast to the *Transition Specification*, which may be derived directly from a model, the *State Reference Chain* is an auxiliary construct that helps to determine the actual internal processes of the state machine implementation, based on given semantical assumptions.

  Accordingly, the chain always needs to end in a simple state, as beyond the transition decomposition from source to target, it also explicitly regards default entries into and exits on completion from composite states. A composite state cannot remain active without activating a contained state[8]. If a transition's target is composite, the chain needs to complement a step into the default state, generally an *Initial* substate (repeat for multiple composition respectively indirect substates).

  To account for the order of exit action execution when traversing hierarchies of states, *Inbound* differentiates the direction of a partial transition into and out of a composite state (note that the flag may change over the course of a transition chain). Here, we find varied precedences for UML statecharts or e.g. STATEMATE[9] (cmp. [HP98]).

- *Guard Condition*
  A boolean function implementing checks and logical comparisons of the variables declared by the *Task Attribute Set*. Note that in an Ada context, a function may not alter variable values; thus, the guard may be considered free from side-effects.

[7]The association is directed from chain to state; the state specification provides no information about actual hierarchy sequences.

[8]We are aware of the fact that this is a semantic variation point in [OMG092], 15.3.11, but consider the alternative impractical.

[9]The focus on UML should not restrict the applicability of our approach to other domains.

The pattern assumes that guards on timing conditions may access a system clock. Alternatively, the HIRTE may dispatch defined timing events as messages to the component queue.

- *Action*
  A procedure aggregating any operations on variables of the *Task Attribute Set.* Additionally, in the context of actions, the HIRTE may allow calls to libraries beyond its sphere of influence, e.g. to interface devices. Restricting these external implementation invocations to action procedures allows limited containment of potential side-effects: address violations are detectable inside the HIRTE space, pre- and postconditions on the attribute set may be checked.

- *Interprete Simple State*
  A procedure that processes a specification record conforming to a simple state. If referenced, first it calls the *Entry Action* procedure.

  After completion, it iterates the *Transitions* list. A semantic variation point offers different solutions to determine a firing transition. The *list order* relates outgoing transitions of the state to transitions of its containing composite states, recursively for further hierarchical containment. Depending on the intended precedence, transitions may be ordered from inner to outer enclosure or vice versa. The *break condition* either concludes the checks after the first positive trigger (accepting the precedence given by the order), or iterates over the full list to resolve other firing criteria the implementation deems expedient, e.g. explicit priorities.

  Determining fire conditions comprises evaluating the guard function and comparing the defined trigger event with the type of message last received. Conditions based on message parameter values would be defined as guards. The trigger check then merges both values (if given). We find another semantic variation point with the mode of event consumption. If the transition specifies both an event and a guard condition, it is possible for a received message to match the event but the guard to circumvent the trigger. In this case, the event may or may not be consumed in the process.

  On leaving the state, i.e. a transition firing, a referenced *Exit Action* executes if indicated. The triggered transition is assigned to the task's *Current Transition* for reference in the outer loop.

- *Interprete Composite State*
  A procedure that processes a specification record conforming to a composite state. As such, conforming to our above statement that any configuration not including a simple state is transitory, it needs to differentiate two cases: an *ingoing* or *outgoing* transition.

Depending on a *true Inbound* flag of the task's *Current Chain*, the configuration increments the state depth and, if referenced, executes the composite states *Entry Action*. On an outbound transition – *Inbound* is *false* –, it decrements the state depth and, depending on the semantic regulation, either calls an *Exit Action*, or pushes it onto a call stack. This call stack would empty on entering a state, thusly marking the outermost exit from some hierarchy.

Before exiting, the procedure assigns the *Next* element to the task's *Current Chain*, effectively providing the outer iteration of the *HIRTE Component* with a further partial transition to process (see above).

4.2.1.4 Implementation

benefits of a generic package

Our Ada reference implementation takes the form of a generic package. This way, it achieves a clean decoupling of variant application-specific structures and the invariant state machine mechanisms. There is an important distinction between this package concept and e.g. a library or a class template: While a common library package would provide the types respectively classes that the state machine specializes and instantiates, the Ada package itself is instantiated for a specific component task. Such a declaration requires a type defining the *State IDs* and the maximum depth of the machine's state hierarchy. The thusly parametrized package represents a blueprint (the types defining the machine's structure) as well as an interpreter for the structures conforming to the *Statically Resolvable State Machine* pattern. In addition to the adapted types necessary to build a machine, it immediately includes attributes required for the processing, instead of leaving them to be declared by the component[10]. The association with a task – i.e. a single caller, no recursion – avoids reentrant interpreter procedures, the package is not shared (cmp. [ISO95], 12.3, par. 13).

The HIRTE state machine spec as given in listing 4.1 uses a package *HIRTE_ Communications* that specifies and implements *Messages*, including their types and buffers, and *Message_ Queues*. HIRTE state machine declarations correspond to the types and attributes of the pattern. Associations between the elements are realized as *access* types. Complementary variables and procedures handle the message queues.

Listing 4.1: HIRTE State Machine Spec

```
with Ada.Real_Time,
      HIRTE_Communications;
use Ada.Real_Time,
     HIRTE_Communications,
     HIRTE_Communications.Messages;
```

[10]This might appear as a deviation from the pattern, but should be considered merely a pragmatic rearrangement, as the pattern is not concernded with implementation specifics like the use of packages, and the package is still instantiated in the context of the component. Same holds true for *Run Machine*.

```

------------------------------- HIRTE State Machine Generics
generic

   type State_ID is private;
   Max_State_Depth : Positive;

package HIRTE_State_Machine is

   type Transition_Specification;
   type Trans_Ref is access all Transition_Specification;
   type Action_Ref is access procedure;
   type Guard_Ref is access function return Boolean;
   type State_Specification;
   type State_Ref_Chain;
   type State_Ref_Chain_Ref is access all State_Ref_Chain;
   type State_Proc_Ref is access procedure;

   type State_Specification is record

      ID : State_ID;
      Executing_Proc : State_Proc_Ref;
      Entry_Action : Action_Ref;
      Exit_Action : Action_Ref;
      Transitions : Trans_Ref;

   end record;

   type State_Ref is access all State_Specification;

   type State_Ref_Chain is record

      Current : State_Ref;
      Inbound : Boolean := True;
      Next : State_Ref_Chain_Ref;

   end record;

   type Transition_Specification is record

      Source : State_ID;
      Target : aliased State_Ref_Chain;
      Event : Messages.Msg_Type;
      Guard : Guard_Ref;
      Action : Action_Ref;
      Next : Trans_Ref;
      -- Add priority if indicated.

   end record;

   type Current_State_Configuration is
     array (1..Max_State_Depth) of State_Ref;

   type Action_Array is
     array (1..Max_State_Depth) of Action_Ref;

   protected type Action_Stack is

      procedure Push(a : in Action_Ref);
      procedure Pop(a : out Action_Ref);
      function Is_Empty return Boolean;

      private

         Put_Index : Integer := 0;
         Actions : Action_Array;

   end;

   -- Attributes for machine processing
   Current_States_Active : Current_State_Configuration;
   Current_Depth : Positive := 1;
   Input_Queues : Message_Queues.Ref_Queue_Ref_List;
   Input_Queue_Count : Integer := 0;
   Output_Queues : Message_Queues.Ref_Queue_Ref_List;
   Output_Queue_Count : Integer := 0;
   Queue_Wait_Delay : Time_Span := Microseconds(500000);
   Current_State : State_Ref;
   Current_Chain : State_Ref_Chain_Ref := null;
   Current_Transition : Trans_Ref := null;
   Transition_Action : Action_Ref := null;
   Last_Message_Type : Messages.Msg_Type;
   Last_Message_Body : Messages.Binary_Buffer;
   Next_Queue_Polled : Integer := 0;
   Next_Queue_Served : Message_Queues.Message_Queue_Index;
   AStack : Action_Stack;

   procedure Get_Message_From_Queues;

   procedure Consume_Last_Message;

   procedure Check_Transition(T : in Trans_Ref;
                              Fire : out Boolean);

   procedure Interprete_Simple_State;
   procedure Interprete_Composite_State;

   procedure Empty_Action_Stack;

   procedure Run_Machine;

end HIRTE_State_Machine;
```

The HIRTE state machine body, listing 4.2, realizes the spec defined above. Variation points annotate implementation alternatives to the solution provided here.

Listing 4.2: HIRTE State Machine Body

```
package body HIRTE_State_Machine is

   procedure Get_Message_From_Queues is
   -- Retrieves a message from the current queue, stores it in
   -- Last_Message_Type and Last_Message_Body. Otherwise switches
   -- to the next queue defined.
   -- Determines the access principle of the machine's queues.

      OK : Boolean := False;

   begin

      Message_Queues.Get_Msg(Input_Queues.all(Next_Queue_Polled).all,
                             Last_Message_Body,
                                  OK);
      if not OK then

         if Next_Queue_Polled+1 > Input_Queue_Count then
            Next_Queue_Polled := 1;
         else
            Next_Queue_Polled := Next_Queue_Polled+1;
         end if;

         delay until Clock + Queue_Wait_Delay;

      else

         Last_Message_Type := Messages.Serialization.
            Get_First_Msg_Byte(Last_Message_Body(1));

      end if;

   end;

   procedure Consume_Last_Message is
   -- "Empties" the last message.
   begin
      Last_Message_Type := Messages.undefined;
   end;

   procedure Check_Transition(T : in Trans_Ref;
                              Fire : out Boolean) is
   -- Determines the trigger conditions (event and guard) of a transition.
   -- Fire is true if event retrieved from queue and guard evaluated to true
   -- (or either undefined), false otherwise.

      Event_Fire,
      Guard_Fire : Boolean := False;

   begin

      if T.Event /= Messages.undefined then

         -- Unconsumed message available? If not, try to retrieve.
         if Last_Message_Type = Messages.undefined then
            Get_Message_From_Queues;
         end if;

         if T.Event = Last_Message_Type then
            -- Variation point: process and compare message
            -- parameters if indicated.
            Event_Fire := True;
         end if;

      else
         Event_Fire := True;
      end if;

      if T.Guard /= null then
         Guard_Fire := T.Guard.all;
      else
         Guard_Fire := True;
      end if;

      Fire := Event_Fire and Guard_Fire;

```

```
end;

procedure Interprete_Simple_State is
-- Calls associated actions, checks transitions.
-- Exits, if a transition fires. Assigns Current_Chain (if transition
-- not null) and Current_Transition (null if final state).

   T_Check : Trans_Ref;
   Firing_Transition : Trans_Ref := null;
   Fire : Boolean := False;

begin

   -- Variation point: depends on hierarchical precedence
   -- of composite state exit actions.
   Empty_Action_Stack;

   Current_States_Active(Current_Depth) := Current_State;

   if Current_State.Entry_Action /= null then
      Current_State.Entry_Action.all;
   end if;

   T_Check := Current_State.Transitions;

   -- Variation point: depends on hierarchical precedence
   -- of regular versus high-level transitions.
   while T_Check /= null loop

      Check_Transition(T_Check, Fire);
      if Fire then
         Firing_Transition := T_Check;
      end if;

      T_Check := T_Check.Next;

      -- Variation point: repeat checks until a transition fires,
      -- or stop for an explicit reinvocation.
      if T_Check = null and Firing_Transition = null then
         T_Check := Current_State.Transitions;
      end if;

      -- Add explicit priorities if indicated.
      -- Add collision detection and warning if indicated.

   end loop;

   if Current_State.Exit_Action /= null then
      Current_State.Exit_Action.all;
   end if;

   if Firing_Transition /= null then

      -- Variation point: consumption of message if event matches
      -- but guard evaluates to false.
      if Firing_Transition.Event /= Messages.undefined
         and Firing_Transition.Event = Last_Message_Type then
         Consume_Last_Message;
      end if;

      Transition_Action := Firing_Transition.Action;
      Current_Chain := Firing_Transition.Target'Access;
```

```

         while Firing_Transition.all.Source
            /= Current_States_Active(Current_Depth).all.ID loop

            -- Resolve high-level transition: leave states until
            -- level is reached.
            Current_Depth := Current_Depth-1;
            if Current_States_Active(Current_Depth).Exit_Action /= null then
               AStack.Push(Current_States_Active(Current_Depth).Exit_Action);
            end if;

         end loop;

      end if;

      Current_Transition := Firing_Transition;

   end;

   procedure Interprete_Composite_State is
   -- Calls associated actions, immediately makes the transition
   -- to a contained (hierarchy increment) or containing (hierarchy
   -- decrement) state. Exits with assigning the next element to
   -- Current_Chain.

   begin

      Current_States_Active(Current_Depth) := Current_State;

      if Current_Chain.Inbound then

         -- Variation point: exit action hierarchy precedence.
         Empty_Action_Stack;

         Current_Depth := Current_Depth+1;

         if Current_State.Entry_Action /= null then
            Current_State.Entry_Action.all;
         end if;

      else -- Outbound

         Current_Depth := Current_Depth-1;

         if Current_State.Exit_Action /= null then
            -- Variation point: exit action hierarchy precedence.
            AStack.Push(Current_State.Exit_Action);
         end if;

      end if;

      Current_Chain := Current_Chain.Next;

   end;

   procedure Empty_Action_Stack is
   -- Pops stacked exit actions and calls them.

      Exit_Action : Action_Ref := null;

   begin

```

```
      if not AStack.Is_Empty then
         loop
            AStack.Pop(Exit_Action);
            exit when Exit_Action = null;
            Exit_Action.all;
         end loop;
      end if;

   end;

   procedure Run_Machine is
   -- Implements an outer loop to iterate the machine's states
   -- until a final state yields a null transition.

   begin

      loop

         -- Variation point: to substitute orthogonal regions,
         -- introduce additional state machine structures.

         Current_State.Executing_Proc.all;

         exit when Current_Transition = null;

         if Transition_Action /= null then
            -- Call a transition action then assign null to
            -- assert it is only called once when traversing
            -- a chained transition.
            Transition_Action.all;
            Transition_Action := null;
         end if;

         Current_State := Current_Chain.Current;

      end loop;

   end;

   protected body Action_Stack is
   -- Plain stack for exit actions.

      procedure Push(a : in Action_Ref) is
      begin
         Put_Index := Put_Index+1;
         Actions(Put_Index) := a;
      end;

      procedure Pop(a : out Action_Ref) is
      begin
         if Put_Index > 0 then
            a := Actions(Put_Index);
            Put_Index := Put_Index-1;
         else
            a := null;
         end if;
      end;

      function Is_Empty return Boolean is
      begin
         return Put_Index = 0;
      end;
```

```

    end;

end HIRTE_State_Machine;
```

multiple structures

The procedure *Run_ Machine* (l. 211) presents a significant variation point with the potential extension toward *n* state machines iterated by the main loop: as an array of size *n*, *Current_ State* then references a sequence of structures to interprete successively. To cope with divergent state complexity between the machines, i.e. one state regularly claims more run-time for processing than another, a simple (static) scheduler may be realized by adapting the call frequency of each *Executing_ Proc* accordingly.

event parameters

Another variation point is given by the optional extension of events by parameters. In addition to the expected message type as referenced in the transition record, the procedure may also compare a sequence of TLV-encoded parameter values with the fields of a received message.

function pointers versus dispatching

Despite the arguments above, one might still argue that attributing the structures with pointers to functions for their processing is just a variant on, or veiled, dispatching. However, there are significant differences. Dispatching is handled by the RTE and requires a corresponding module to manage the assignment during run-time that is not (without effort) influencable by the developer. Function pointers on the other hand are assigned explicitly by the program, without any additional RTE support. In contrast, dispatching is a complex, black-box mechanic, while pointer assignment and evaluation are (very) basic, side-effect free operations.

4.2.1.5 Instantiation and Application

Consider the state machine illustrated by the statechart in figure 4.6, assume it is associated with the behavior of a *Component A*. To implement it conforming to the pattern, we first describe the specification with listing 4.3.

Listing 4.3: Pattern-conforming Example Spec

```
package Component_A is

    -- States ...
    type Component_A_State_ID is (Initial, Simple_1, Simple_2, Simple_3,
        Composite_1, Composite_2, Final, Composite_1_Initial,
        Composite_1_Sub_1, Composite_1_Sub_2, Composite_1_Final,
        Composite_2_Initial, Composite_2_Sub_1, Composite_2_Sub_2,
        Composite_2_Final);

    -- ... and their numerical representation (explicit, for the purpose of
    --      state tracing by other components).
    for Component_A_State_ID use (Initial => 1, Simple_1 => 2,
        Simple_2 => 3, Simple_3 => 4, Composite_1 => 5, Composite_2 => 6,
        Final => 7, Composite_1_Initial => 8, Composite_1_Sub_1 => 9,
        Composite_1_Sub_2 => 10, Composite_1_Final => 11,
        Composite_2_Initial => 12, Composite_2_Sub_1 => 13,
        Composite_2_Sub_2 => 14, Composite_2_Final => 15);

```

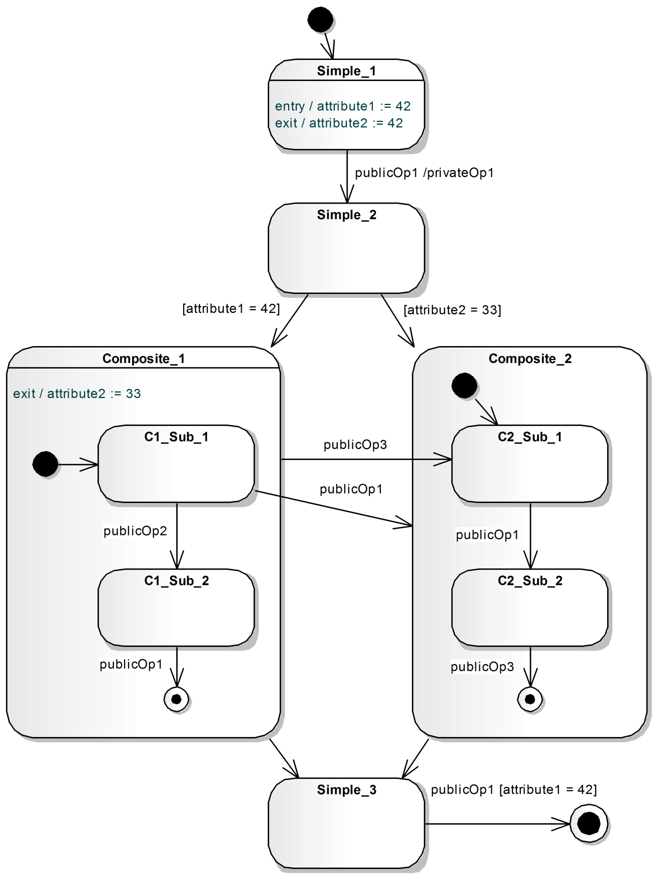


Figure 4.6: Pattern Example Statechart

```
-- Same for transitions.
type Component_A_Transition_ID is (T_Initial_TO_Simple_1,
    T_Simple_1_TO_Simple_2, T_Simple_2_TO_Composite_1,
    T_Simple_2_TO_Composite_2, T_Composite_1_TO_Composite_2_Sub_1,
    T_Composite_1_TO_Simple_3, T_Composite_1_Initial_TO_Composite_1_Sub_1,
    T_Composite_1_Sub_1_TO_Composite_1_Sub_2,
    T_Composite_1_Sub_1_TO_Composite_2,
    T_Composite_1_Sub_2_TO_Composite_1_Final, T_Composite_2_TO_Simple_3,
    T_Composite_2_Initial_TO_Composite_2_Sub_1,
    T_Composite_2_Sub_1_TO_Composite_2_Sub_2,
    T_Composite_2_Sub_2_TO_Composite_2_Final, T_Simple_3_TO_Final);

for Component_A_Transition_ID use (T_Initial_TO_Simple_1 => 1,
    T_Simple_1_TO_Simple_2 => 2, T_Simple_2_TO_Composite_1 => 3,
    T_Simple_2_TO_Composite_2 => 4,
    T_Composite_1_TO_Composite_2_Sub_1 => 5,
    T_Composite_1_TO_Simple_3 => 6,
    T_Composite_1_Initial_TO_Composite_1_Sub_1 => 7,
    T_Composite_1_Sub_1_TO_Composite_1_Sub_2 => 8,
    T_Composite_1_Sub_1_TO_Composite_2 => 9,
    T_Composite_1_Sub_2_TO_Composite_1_Final => 10,
      T_Composite_2_TO_Simple_3 => 11,
      T_Composite_2_Initial_TO_Composite_2_Sub_1 => 12,
      T_Composite_2_Sub_1_TO_Composite_2_Sub_2 => 13,
      T_Composite_2_Sub_2_TO_Composite_2_Final => 14,
      T_Simple_3_TO_Final => 15);

   -- Instantiate the state machine package to obtain a specialized copy:
   package Component_A_State_Machine is
     new HIRTE_State_Machine(State_ID => Component_A_State_ID,
        Max_State_Depth => 2);
   use Component_A_State_Machine;

   -- To store the current state configuration:
   type State_Array is array (1..2) of Component_A_State_ID;

   -- The attribute set of Component A:
   type Component_A_Attribute_Set is record

      Current_States_Active : State_Array;
      attribute1 : Integer;
      attribute2 : Integer;

   end record;

   Attributes : Component_A_Attribute_Set;

   -- Actions:
   procedure A_privateOp1;
   procedure A_Simple_1_Entry;
   procedure A_Simple_1_Exit;
   procedure Composite_1_Exit;

   -- Guards:
   function Guard_Simple_2_TO_Composite_1 return Boolean;
   function Guard_Simple_2_TO_Composite_2 return Boolean;
   function Guard_Simple_3_TO_Final return Boolean;
   function Guard_Composite_1_TO_Simple_3 return Boolean;
   function Guard_Composite_2_TO_Simple_3 return Boolean;

   -- Arrays to construct and configure the state machine:
   Component_A_States : array (Initial..Composite_2_Final)
      of aliased Component_A_State_Machine.State_Specification;
   Component_A_Transitions : array (T_Initial_TO_Simple_1..T_Simple_3_TO_Final)
      of aliased Component_A_State_Machine.Transition_Specification;

   -- Arrays to configure the transition chains:
   Simple_2_TO_Composite_1_Chain : array (1..1)
      of aliased Component_A_State_Machine.State_Ref_Chain;
   Simple_2_TO_Composite_2_Chain : array (1..1)
      of aliased Component_A_State_Machine.State_Ref_Chain;
   Composite_1_Sub_1_TO_Composite_2_Chain : array (1..2)
      of aliased Component_A_State_Machine.State_Ref_Chain;
   Composite_1_TO_Composite_2_Sub_1_Chain : array (1..1)
      of aliased Component_A_State_Machine.State_Ref_Chain;

   -- A dedicated procedure to handle all pointer assignments:
   procedure Construct_State_Machine;

   -- The actual component task with communication queues:
   task type Component_A_Task (
      To_Kernel_Queue : Message_Queues.Ref_Queue_Ref_List;
      From_Kernel_Queue : Message_Queues.Ref_Queue_Ref_List) is
   end Component_A_Task;

end Component_A;
```

programming and code generation

Aside from providing a scheme to apply the introduced pattern to an actual state machine, the spec conveys an inherent formalism for both systematic manual programming and model transformation. Hence e.g. the one-element-arrays for transition chains (l. 86) that a developer would probably recast to plain variables, but which otherwise reflect the underlying chain construction principle *array size = number of composite state borders crossed.* This mechanistic approach is continued in the body of listing 4.4. As the underlying heuristic is manifest (see below), the source is truncated accordingly.

Listing 4.4: Pattern-conforming Example Body

```
package body Component_A is

   procedure A_privateOp1 is
   begin
      -- Some operation on the task attribute set.
   end;

   procedure A_Simple_1_Entry is
   begin
      Attributes.attribute1 := 42;
   end;
   ...

   function Guard_Simple_2_TO_Composite_1 return Boolean is
   begin
      if Attributes.attribute1 = 42 then
         return True;
      else
         return False;
      end if;
   end;
   ...

   function Guard_Composite_1_TO_Simple_3 return Boolean is
   -- Completion condition for Composite_1 region.
   begin
      if Component_A_State_Machine.Current_States_Active(2).ID
         = Composite_1_Final then
         return True;
      else
         return False;
      end if;
   end;
   ...

   procedure Construct_State_Machine is
   begin

      -- Attribute transitions:

      Component_A_Transitions(T_Initial_TO_Simple_1).Source := Initial;
      Component_A_Transitions(T_Initial_TO_Simple_1).Target.Current :=
         Component_A_States(Simple_1)'Access;
      Component_A_Transitions(T_Initial_TO_Simple_1).Target.Next := null;
```

```
    Component_A_Transitions(T_Initial_TO_Simple_1).Event :=
        Messages.undefined;
    Component_A_Transitions(T_Initial_TO_Simple_1).Guard := null;
    Component_A_Transitions(T_Initial_TO_Simple_1).Action := null;
    Component_A_Transitions(T_Initial_TO_Simple_1).Next := null;

    Component_A_Transitions(T_Simple_1_TO_Simple_2).Source := Simple_1;
    Component_A_Transitions(T_Simple_1_TO_Simple_2).Target.Current :=
        Component_A_States(Simple_2)'Access;
    Component_A_Transitions(T_Simple_1_TO_Simple_2).Target.Next := null;
    Component_A_Transitions(T_Simple_1_TO_Simple_2).Event :=
        Messages.publicOp1;
    Component_A_Transitions(T_Simple_1_TO_Simple_2).Guard := null;
    Component_A_Transitions(T_Simple_1_TO_Simple_2).Action :=
        A_privateOp1'Access;
    Component_A_Transitions(T_Simple_1_TO_Simple_2).Next := null;
    ...

    Component_A_Transitions(T_Composite_1_Sub_1_TO_Composite_2).Source :=
        Composite_1_Sub_1;
    Component_A_Transitions(T_Composite_1_Sub_1_TO_Composite_2).Target.
        Current := Component_A_States(Composite_1)'Access;
    Component_A_Transitions(T_Composite_1_Sub_1_TO_Composite_2).Target.
        Inbound := False;
    Component_A_Transitions(T_Composite_1_Sub_1_TO_Composite_2).Target.
        Next := Composite_1_Sub_1_TO_Composite_2_Chain(1)'Access;
    Component_A_Transitions(T_Composite_1_Sub_1_TO_Composite_2).Event :=
        Messages.publicOp1;
    Component_A_Transitions(T_Composite_1_Sub_1_TO_Composite_2).Guard := null;
    Component_A_Transitions(T_Composite_1_Sub_1_TO_Composite_2).Action := null;
    Component_A_Transitions(T_Composite_1_Sub_1_TO_Composite_2).Next :=
        Component_A_Transitions(T_Composite_1_TO_Composite_2_Sub_1)'Access;
    ...

    -- Attribute states:

    Component_A_States(Initial).ID := Initial;
    Component_A_States(Initial).Executing_Proc :=
        Component_A_State_Machine.Interprete_Simple_State'Access;
    Component_A_States(Initial).Entry_Action := null;
    Component_A_States(Initial).Exit_Action := null;
    Component_A_States(Initial).Transitions :=
        Component_A_Transitions(T_Initial_TO_Simple_1)'Access;

    Component_A_States(Simple_1).ID := Simple_1;
    Component_A_States(Simple_1).Executing_Proc :=
        Component_A_State_Machine.Interprete_Simple_State'Access;
    Component_A_States(Simple_1).Entry_Action := A_Simple_1_Entry'Access;
    Component_A_States(Simple_1).Exit_Action := A_Simple_1_Exit'Access;
    Component_A_States(Simple_1).Transitions :=
        Component_A_Transitions(T_Simple_1_TO_Simple_2)'Access;
    ...

    Component_A_States(Composite_1).ID := Composite_1;
    Component_A_States(Composite_1).Executing_Proc :=
        Component_A_State_Machine.Interprete_Composite_State'Access;
    Component_A_States(Composite_1).Entry_Action := null;
    Component_A_States(Composite_1).Exit_Action := Composite_1_Exit'Access;
    Component_A_States(Composite_1).Transitions :=
        Component_A_Transitions(T_Composite_1_TO_Composite_2_Sub_1)'Access;
    ...

```

```
      Component_A_States(Final).ID := Final;
      Component_A_States(Final).Executing_Proc :=
         Component_A_State_Machine.Interprete_Simple_State'Access;
      Component_A_States(Final).Entry_Action := null;
      Component_A_States(Final).Exit_Action := null;
      Component_A_States(Final).Transitions := null;
      ...

      -- Attribute transition state chains:

        Simple_2_TO_Composite_1_Chain(1).Current :=
           Component_A_States(Composite_1_Initial)'Access;
        Simple_2_TO_Composite_1_Chain(1).Next := null;

        Simple_2_TO_Composite_2_Chain(1).Current :=
           Component_A_States(Composite_2_Initial)'Access;
        Simple_2_TO_Composite_2_Chain(1).Next := null;

      Composite_1_Sub_1_TO_Composite_2_Chain(1).Current :=
         Component_A_States(Composite_2)'Access;
      Composite_1_Sub_1_TO_Composite_2_Chain(1).Next :=
         Composite_1_Sub_1_TO_Composite_2_Chain(2)'Access;
      Composite_1_Sub_1_TO_Composite_2_Chain(2).Current :=
         Component_A_States(Composite_2_Initial)'Access;
      Composite_1_Sub_1_TO_Composite_2_Chain(2).Next := null;

        Composite_1_TO_Composite_2_Sub_1_Chain(1).Current :=
           Component_A_States(Composite_2_Sub_1)'Access;
        Composite_1_TO_Composite_2_Sub_1_Chain(1).Next := null;

   end;

   task body Component_A_Task is
   begin

      Component_A_State_Machine.Current_States_Active(1) :=
         Component_A_States(Initial)'Access;
      Construct_State_Machine;
      Attributes.attribute1 := 0;
      Attributes.attribute2 := 0;

      Component_A_State_Machine.Last_Message_Type := Messages.undefined;
      Component_A_State_Machine.Input_Queues := From_Kernel_Queue;
      Component_A_State_Machine.Input_Queue_Count := 1;
      Component_A_State_Machine.Next_Queue_Polled := 1;
      Component_A_State_Machine.Next_Queue_Served := 1;
      Component_A_State_Machine.Current_State :=
         Component_A_States(Initial)'Access;
      Component_A_State_Machine.Current_Chain := null;
      Component_A_State_Machine.Current_Transition := null;

      Component_A_State_Machine.Run_Machine;

   end Component_A_Task;

end Component_A;
```

For the instantiation and application of the *Statically Resolvable State Machine Pattern*, we summarize the following rules:

1. *Completion conditions* of regions are complemented as guard condition functions. The array index for the current state configuration is given by the hierarchy level of the corresponding composite state.

2. *Access types, addresses or pointers* reference statically declared elements.

3. The procedure *Construct_ State_ Machine* handles all initial pointer assignments constituting the state machine structure. All subsequent pointer operations merely reference the established structure and its elements, e.g. for transition chain iterations.

4. Conditionless transitions from *initial states* are attributed with *null* for guards and events.

5. *Outgoing transitions* of a state are linked by the *Next* attribute. The order is arbitrary. *Next = null* ends the list.

6. *High-level transitions* are assigned to the *Next* attribute of the last element of the transition list, successively for each inner-to-outer hierarchy level of containing composite states (note the variation point in the sources above).

7. A *transition chain* is attributed whenever a transition crosses the borders of composite states, explicitly or implicitly (e.g. via a region's initial state). Each element represents, i.e. references, a visited state, ending with the target, *Next = null* concluding the list. The *Target* of a transition that references a chain with *Target.Next* is the composite state visited first in the chain, i.e. the chain itself's first *Current* reference would be the second step. "Visited" applies to inbound or outgoing transition steps, distinguished by *Inbound* for each step. This direction flag may change over the course of a chain, conforming to a transition from a contained state to another state contained in a different region.

8. A *state's transition reference* is assigned the first element of the outgoing transition list (see above).

9. The state machine's *final state* is assigned *null* as transition reference.

4.2.1.6 Eliminating Pointers

pointers vs. IDs

In a scenario that completely restricts the use of pointers of any kind, e.g. a SPARK ([Ba03]) implementation, we need to substitute the references as realized above. A valid and immediate approach is presented by the already defined *State ID* and *Transition ID* types and the corresponding arrays.

pointers and decomposition

The difference, and an important reason why we relied on pointers in the reference implementation, is that the decomposition into generic state machine mechanisms and actual application structures is countermanded: the state machine records and procedures need the IDs as array indices to address the state and transition instances. In consequence, the *HIRTE_ State_ Machine* package would now depend on the package *Component_ A*, resulting in a cyclic relationship.

merging generic and application package

We can resolve this by merging both packages, integrating the type definitions and instantiations. The result is lacking the generic setup of the reference sources, but retains all significant qualities – static resolvability and efficient processing. Along the way, another requirement of SPARK, exclusion of templates, is met.

4.2.2 State Tracer Pattern

4.2.2.1 Abstract

First and foremost, the *State Tracer* is a direct response to requirements III (activity transparency) and VII (activity persistency). On a more general level, it provides a solution for immediate, quasi-RT logging. Common approaches to implement a logging component define an interface with mutual exclusive write operations, respectively a queue for log entries, that controls access to one or more log files. Due to this organization, some delay of the state report is inevitable. The tracer instead conceives a sequence of n memory segments for n components realizing services to directly register their current states, collision-free. These state data arrays are compiled to a state configuration trace in a defined frequency or triggered by events.

4.2.2.2 Structure

The structure of the pattern is described by fig. 4.7.

4.2.2.3 Collaboration

- *Service Component*
  An active *task*, implementing some service of the application. The component defines a method *logState* that takes the current component state to encode and store it as byte array. For the states, an enumeration *E-States* specifies the corresponding byte values, therefore also determining the set of traceable states in the context of a given application. Additionally, the array may include any variable values necessary to represent the complete state of the service.

  Intuitively, the log method might be provided by the associated *Current State Log*. The departure from established OO conventions allows a more direct, efficient write operation: instead of referencing an encapsuled log object, *stateLog* points to a memory segment that is exclusively allocated for one service component. In consequence, the implementation can avoid mutex or reentrancy overhead. Another potential problem

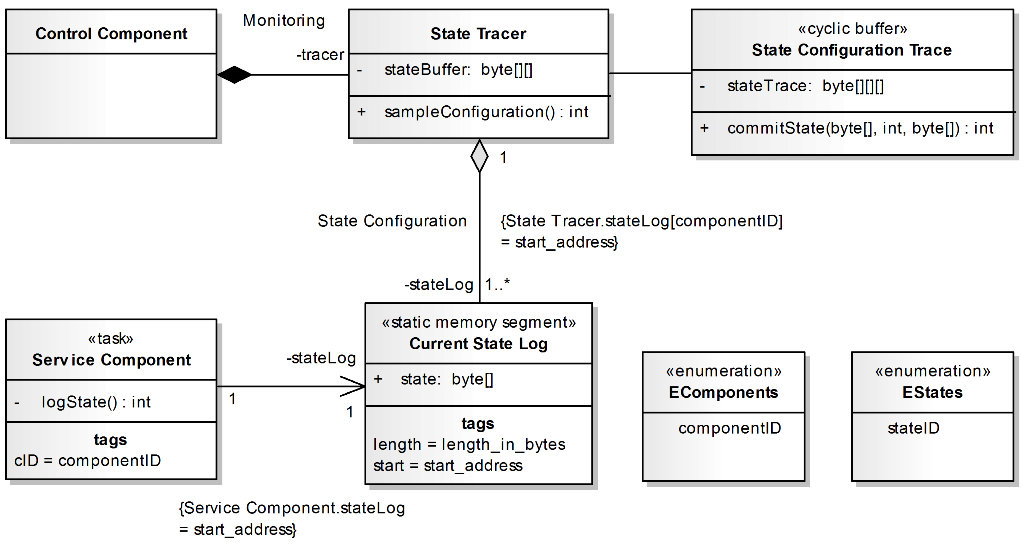


Figure 4.7: State Tracer Pattern Structure

(as identified in 3.2.3) is tackled by eliminating the need for generic, dynamically bound parameters, respectively a polymorphic log method that would be necessary to handle the diverse state-related component attributes.

- *Current State Log*
  A *static memory segment* stores the current state of the associated component as binary array. I.e. on concluding a transition, the component calls *logState*, writing a fixed *length* segment to the *start_ address*.

- *State Tracer*
  A passive module that aggregates the current state logs of all components, identified and referenced by the enumeration *EComponents*, and provides an intermediate *stateBuffer* for a complete configuration, i.e. the states of all components for a given instant, or time *t*. This sample is assembled by *sampleConfiguration*, which copies the content of all logs from their start addresses to the transient buffer, clearing the log segments for overwriting. Subsequently, the state configuration is committed to the associated *State Configuration Trace*.

  Depending on the complexity and frequency of entries, the *stateBuffer* itself may be realized as cyclic buffer to maintain timing constraints between RT logging and the (long-term) trace archive.

- *State Configuration Trace*
  A *cyclic buffer* used by the tracer to store state configurations of *n* steps

(see below). *commitState* accepts a state array, a step number to assure a relative order and component ID as parameters. The cyclic organization implies that after writing the *n*th step, the first entry will be overwritten. In practice however, the sequence of stored steps may not necessarily be strictly successive, as certain conditions may forbid overwriting of critical entries, e.g. fault states or states that hint at manipulation.

- *Control Component*
  An active kernel module that composes the *State Tracer* for monitoring the application's components. It triggers the sampling by the tracer, thus effectively defining a *step* of the trace. This invocation may be time- or event-based. In the time-based case, the interval and frequency of samples needs to consider the transition frequency of the traced components in order to gain meaningful data while minimizing log memory consumption. The event-based case can introduce a feedback loop: if e.g. the control component imposes step semantics on the components, it is able to register completion (all traced components processed their current states/transitions). Before initiating the next step, the state configuration is sampled.

4.2.2.4 Implementation

To integrate a state tracer into the architecture, we extend the component package described in 4.3 by the structures as given in listing 4.5.

Listing 4.5: Extended Component Package Spec

```
package Component_A is

   ...

   type Component_A_Attribute_Set is record

      Component_ID : State_Tracer.EComponents;
      Current_States_Active : State_Array;
      attribute1 : Integer;
      attribute2 : Integer;
      Last_Message_Type : Messages.Msg_Type;
      Last_Message_Body : Messages.Binary_Buffer;

   end record;

   for Component_A_Attribute_Set use record

      Component_ID at 0 range 0..7;
      Current_States_Active at 1 range 0..15;
      attribute1 at 3 range 0..31;
      attribute2 at 8 range 0..31;
      Last_Message_Type at 12 range 0..7;
      Last_Message_Body at 13 range 0..MSG_BUFFER_SIZE_BITS-1;

   end record;

   for Component_A_Attribute_Set'Size use 152;
```

```
28
29     package Serialization is
30
31        package Component_A_Attribute_Set_Serialization is
32          new Ada.Storage_IO(Component_A_Attribute_Set);
33
34     end Serialization;
35
36     procedure Log_State(Attribute_Set : in Component_A_Attribute_Set;
37                         State_Log : out Storage_Array);
38
39     task type Component_A_Task (
40          To_Kernel_Queue : Message_Queues.Ref_Queue_Ref_List;
41          From_Kernel_Queue : Message_Queues.Ref_Queue_Ref_List;
42          State_Log : HIRTE.State_Tracer.Component_A_Storage_Array_Ref) is
43     end Component_A_Task;
44
45  end Component_A;
```

The task attribute set is complemented by two additional entries (l. 5; a component identification, the recent message) and a specification of its binary encoding, a *record representation clause* (l. 16), resulting in an array of 152/8 = 19 *Storage Elements*, i.e. bytes. An instantiation[11] of *Ada.Storage_IO* for the defined type provides functions for de-/serialization of the set. Conforming to the pattern, the component package includes a *Log_State* procedure to serialize a given state into a log storage. The component task definition itself adds a parameter referencing the dedicated log memory.

Listing 4.6: *Log_State* Body

```
package body Component_A is

   procedure Log_State(Attribute_Set : in Component_A_Attribute_Set;
                       State_Log : out Storage_Array) is
      begin

         Component_A.Serialization.
            Component_A_Attribute_Set_Serialization.
               Write(State_Log, Attributes);

      end;

      ...
```

The *Log_State* implementation is described by listing 4.6: application of the previously declared package *Component_A.Serialization* writes the binary attribute set representation to the provided storage array.

Listing 4.7 defines the state tracer specification.

Listing 4.7: State Tracer Spec

```
package State_Tracer is

   STATE_CONFIGURATION_TRACE_ENTRIES : constant Positive := 100;
```

[11]Despite of the *new* allocator, this expression does not violate the imposed restrictions. Instantiation happens at compile-time.

```
TRACE_STEP_RANGE : constant Positive := 100000;
STATE_BYTE_LENGTH : constant Storage_Count := 1;
COMPONENT_ID_BYTE_LENGTH : constant Storage_Count := 1;
KERNEL_STATE_LOG_LENGTH : constant Storage_Count := 177/8;
COMPONENT_A_STATE_LOG_LENGTH : constant Storage_Count := 152/8;
COMPONENT_B_STATE_LOG_LENGTH : constant Storage_Count := 150/8;
STATE_TRACER_ENTRY_LENGTH : constant Storage_Count :=
   KERNEL_STATE_LOG_LENGTH
   + COMPONENT_A_STATE_LOG_LENGTH
   + COMPONENT_B_STATE_LOG_LENGTH;
KERNEL_LOG_OFFSET : constant Storage_Count := 0;
COMPONENT_A_LOG_OFFSET : constant Storage_Count :=
   KERNEL_LOG_OFFSET + KERNEL_STATE_LOG_LENGTH;
COMPONENT_B_LOG_OFFSET : constant Storage_Count :=
   COMPONENT_A_LOG_OFFSET + COMPONENT_A_STATE_LOG_LENGTH;

type EComponents is (kernel, component_a, component_b);
for EComponents'Size use 8;
for EComponents use (kernel => 1, component_a => 2,
                     component_b => 3);

subtype Current_State_Log is
   Storage_Array(1..STATE_TRACER_ENTRY_LENGTH);
type Current_State_Log_Ref is access all Current_State_Log;
type Kernel_Storage_Array_Ref is
   access all Storage_Array(1..KERNEL_STATE_LOG_LENGTH);
type Component_A_Storage_Array_Ref is
   access all Storage_Array(1..COMPONENT_A_STATE_LOG_LENGTH);
type Component_B_Storage_Array_Ref is
   access all Storage_Array(1..COMPONENT_B_STATE_LOG_LENGTH);
type Trace_Step_Sequence is mod TRACE_STEP_RANGE;
type Trace_Entry_Index is mod STATE_CONFIGURATION_TRACE_ENTRIES;
type Entry_Priority is (critical, warning, regular, info);
for Entry_Priority'Size use 2;

type State_Trace_Entry is record

   Step : Trace_Step_Sequence;
   Priority : Entry_Priority;
   Timestamp : Time_Span;
   State_Trace : Current_State_Log;

end record;

type State_Trace_Array is
   array (Trace_Entry_Index) of State_Trace_Entry;

protected type State_Configuration_Trace is

   procedure Commit_State(Step : in Trace_Step_Sequence;
                          Priority : in Entry_Priority;
                          Timestamp: in Time_Span;
                          State : in Current_State_Log;
                          Collisions : out Integer);

   procedure Print_Trace;

private

   Put_Index : Trace_Entry_Index := 0;
   State_Trace : State_Trace_Array;

```

```
66     end;
67
68     type State_Configuration_Trace_Ref is
69        access all State_Configuration_Trace;
70
71     type Log_Address_Array is
72        array (1..NO_OF_COMPONENTS) of Storage_Count;
73
74     protected type State_Tracer(Log_Reference : Current_State_Log_Ref;
75                                 Trace_Reference : State_Configuration_Trace_Ref) is
76
77        procedure Sample_Configuration(Sample_Status : out Integer);
78
79     private
80
81        Current_Step : Trace_Step_Sequence := 1;
82        State_Buffer : State_Trace_Entry;
83        Component_Log_Address : Log_Address_Array :=
84           (KERNEL_LOG_OFFSET, COMPONENT_A_LOG_OFFSET);
85
86     end;
87
88     Type State_Tracer_Ref is access all State_Tracer;
89
90     Initial_Time : Time := Clock;
91
92  end State_Tracer;
```

The specification defines a sequence of constants that determine the number and size of state log entries, range of the step counter and relative addresses of each component log in the complete storage array. The trace step range is required to be larger than the entry count to maintain the invariant of an intact relative order.

A component ID enumeration (l. 20) is declared in the context of the state tracer, as it is generally associated with component control. The subtype *Current_ State_ Log* corresponds to the pattern element of the same name, with the subsequent declarations mapping the sections dedicated to components (in the given example *Kernel* and *Component A*). By defining the trace step sequence and trace entry index as *modulo* types, they implement a wrap-around arithmetic, conforming to the relative order invariant and cyclic buffer, respectively. For illustration purposes, an example of four entry priorities (l. 36; encoded in two bits) is given – they may be used to prevent overwriting of log entries or realize automatic alerts in critical states.

The pattern elements *State Tracer* and *State Configuration Trace* are implemented as *protected types*, their bodies realized in listing 4.8. With the *Component_ Log_ Address* array (l. 83), the *State_ Tracer* gains a handle on the set of all aggregated component logs. In this reference implementation, this is less crucial, as all sublogs are contained in one storage array. Another implementation however may distribute the individual logs arbitrarily and incoherently.

Listing 4.8: State Tracer and Configuration Trace Bodies

```
package body State_Tracer is

  protected body State_Configuration_Trace is

     procedure Commit_State(Step : in Trace_Step_Sequence;
                            Priority : in Entry_Priority;
                            Timestamp: in Time_Span;
                            State : in Current_State_Log;
                            Collisions : out Integer) is
     begin
        State_Trace(Put_Index).Step := Step;
        State_Trace(Put_Index).Priority := Priority;
        State_Trace(Put_Index).Timestamp := Timestamp;
        State_Trace(Put_Index).State_Trace := State;
        -- Implement checks for critical priority entries here,
        -- if indicated.
        Collisions := 0;
        Put_Index := Put_Index+1;
     end;

  end;

  protected body State_Tracer is

     procedure Sample_Configuration(Sample_Status : out Integer) is
        Collision : Integer;
     begin
        State_Buffer.Step := Current_Step;
        State_Buffer.Timestamp := Clock-Initial_Time;
        State_Buffer.State_Trace := Log_Reference.all;
        Trace_Reference.all.Commit_State(State_Buffer.Step, regular,
           State_Buffer.Timestamp, State_Buffer.State_Trace, Collision);
        Current_Step := Current_Step+1;
        -- Implement reactions on critical entry collisions here,
        -- if indicated.
        Sample_Status := 1;
     end;

  end;

end State_Tracer;
```

Both procedures given here are kept most rudimentary. The tracer commits each entry directly after taking the sample.

Based on these specs, the tracer is instantiated and referenced in the environment (listing 4.9).

Listing 4.9: Instantiating the Tracer

```
package HIRTE_Environment is

    ...

    State_Log : aliased HIRTE.State_Tracer.Current_State_Log;

    HIRTE_Kernel_State_Log :
       aliased Storage_Array(1..HIRTE.State_Tracer.
          KERNEL_STATE_LOG_LENGTH);
    for HIRTE_Kernel_State_Log'Address use
       State_Log'Address+HIRTE.State_Tracer.KERNEL_LOG_OFFSET;

```

```
13     HIRTE_Component_A_State_Log :
14        aliased Storage_Array(1..HIRTE.State_Tracer.
15           COMPONENT_A_STATE_LOG_LENGTH);
16     for HIRTE_Component_A_State_Log'Address use
17        State_Log'Address+HIRTE.State_Tracer.COMPONENT_A_LOG_OFFSET;
18
19     The_State_Config_Trace : aliased HIRTE.State_Tracer.
20        State_Configuration_Trace;
21     The_State_Tracer : aliased HIRTE.State_Tracer.
22        State_Tracer(State_Log'Access, a_State_Config_Trace'Access);
23
24     The_HIRTE_Kernel : HIRTE.Kernel.Kernel_Task(
25                                 HIRTE_Kernel_Input'Access,
26                                 HIRTE_Kernel_Output'Access,
27                                 HIRTE_Kernel_State_Log'Access,
28                                 The_State_Tracer'Access);
29
30     HIRTE_Component_A : HIRTE.Component_A.Component_A_Task(
31                                 Queue_C_A_to_K'Access,
32                                 Queue_K_to_C_A'Access,
33                                 HIRTE_Component_A_State_Log'Access);
34
35  end HIRTE_Environment;
```

The environment uses the tracer spec's constants to complement the sizes of the declarations. In the given implementation, we declare one coherent *State_Log* as container for all component logs. After declaring a component sublog, the address of its *Storage_Array* is set to a section of the composing log, based on an offset (ls. 10, 16). Conforming to the pattern approach, this address may be any other fitting memory segment. In that case, instead of passing the state tracer instance a single log reference, it would require a set.

4.2.3 Virtual Control Unit Pattern

4.2.3.1 Abstract

Chapter 2.1.1 described the *Electronic Control Units* of the automotive domain. In this section, we combine the notion of such delimited components controlled by restrictive programs with a refinement respectively specialization of virtual machines (cmp. the virtual machine pattern in [Do03], 4.7) to *Virtual Control Units* (VCU). As an additional constraint, our requirement II (system state automata) constitutes the build of a control unit program, consequently designated *Interpretable State Machine Code* (ISMC).

The resulting VCU represents a lightweight, state automaton-driven virtual machine. It is designed

- to be *scalable* and *adaptable* – operations, i.e. actions and guards, may be implemented *natively* as interpreter command sequences or *externally* in corresponding operation sets. Both options need to conform to certain restrictions (see next sections), but aside from that may realize arbitrarily simple or complex logics of any kind. With the VCUs providing a safe RTE, their composing application might represent a slim solution (cmp.

2.2.3), its operations merely reading out sensors and handling OTA data distribution, or a fat client, with computation-heavy map-matching and database operations. By keeping the essential VCU mechanisms generic, we gain the additional benefit of adaptability to other telematics domains by replacing the operations.

- to be *distributable* – the above scalability directly impacts the resulting executables memory footprints. This implies being able to adapt an application's components to the capabilities of various RTE hardware platforms, and thusly distributing them over a network of e.g. ECUs, OBE and smartphones etc. depending on given economical or technological standards. Interaction with the VCU machine can be completely limited to input and output message queues that in turn may be mapped to any protocol stack from automotive busses to TCP/IP, enabling integration into local (vehicle) and/or OTA (CN) networks.

- to be *externally controllable* – which presents an active scheduling option and a passive view in terms of monitoring. A VCU machine can run unimpeded as a "black box" in the loop of some execution unit, exhibiting similar outside behavior as a component based on the SRSM pattern above. Construction of the interpreter machine given below, however, permits a detailed active control of the ISMC processing steps. Thus, a control entity can explicity schedule execution of a set of VCUs. Passively, in each step, a controlling instance can gain the "white box" state configurations of the interpreter automaton as well as the application task realized by the ISMC program.

divide and conquer

The VCU approach is an example for the pursuit of plain *divide and conquer* deconstruction. Instead of a complex, heavyweight universal virtual machine with dynamic threads and a full language set for programming, we propose a network of strictly controlled lightweight machines, specializable and statically (safely) implemented, and a limited (manageable) language that facilitates transparency of behavior. In this regard, the above ECU analogy emphasizes the distinction between a VCU and e.g. a JavaVM: as an ECU with its limited but deterministic capabilities is deployed in a risk-averse (sub-)system, reciprocally, a complex VM, if anything with nondeterministic features like garbage collection, is unsuited for safety-critical scenarios that require full white-box reproducibility of activities.

4.2.3.2 Interpretable State Machine Code

origin of ISMC

In 3.1.2 we established the usefulness of state automata in the operation of complex distributed systems. As an answer, 4.2.1 proposed a pattern to build components that adhere to state machine behavior. Further development of this concept now leads us to a program scheme that not only mimics but structurally represents a state automaton. For that purpose, this section defines a language which allows the description of state automata with (in the given case) binary terminals – the *Interpretable State Machine Code*[12].

designation of ISMC

While any format could be chosen for convenience, the binary terminals consciously emphasize the disregard for options of some intermediate format, like XML. The resulting byte sequences are meant to represent machine programs, i.e. compact instructions for a RTE. They need to be efficiently interpretable, implying an alignment toward optimized parsing for execution, in contrast to exhaustive structural exploration as found e.g. in the XMI DTD for the UML metamodel ([OMG07]). Furthermore, the concept has to cope with the memory constraints of embedded devices and OTA interfaces the given domain imposes on any realization. Thus, we prefer a compact format from the start, optimizable in the process (cmp. 4.2.3.6).

Turing machines inspiring ISMC

The idea of our specific state machine interpreter approach is loosely borrowed from the concept of *Turing machines*[13] (cmp. [Co04] for a compilation of the papers). To begin with, a common notion of the derived requirements I to VI is *controllability* – clear-cut components with deterministic or at least fully reproducible behavior. In the context of a "safe application in an unsafe environment", a consistent implication is to consider a proprietary RTE. This still does not guarantee faultless execution[14], but it goes a long way toward avoiding side-effects and ensuring monitoring of our own applications. An additional benefit is a general independence from existing (or lacking) RTEs. Join requirement VII: persistent transitions of the complete state configuration are greatly facilitated by containing the run-time representation of an application in a coherent repository.

setup of an ISMC program

This leads us to the general setup as illustrated by fig. 4.8. Due to its intended closed character, ISMC complements the automaton structures – the actual program with *State 1* to *n* – with segments that provide initial information and run-time registers for the interpreter (notably including the *interpreter state register* and *head registers*), and working memory for the implemented application. Thus, this *tape* contains the complete component program state at any time *t* during execution.

setup of the interpreter

The interpreter is realized as a state machine itself; its *transition function.* Based on the instructions read from the tape's *cells* at the positions indicated by the *interpreter head*, *transition chain head* and of the registers, it traverses the states and transitions. To reduce complexity of the interpreter machine and encapsulate specifics of the actual binary code, a *tape reader* compiles tokens from a sequence of cells and passes them, along with their types, to the

[12]In the following, we use the term "ISMC" for both the language format/type and an instance, i.e. a program, byte sequence or machine represented in ISMC.

[13]We are aware of the fact that they are meant as a vehicle for theoretical considerations. Still, their setup also is useful practically.

[14]Obviously, if the environment is not under control, what could?

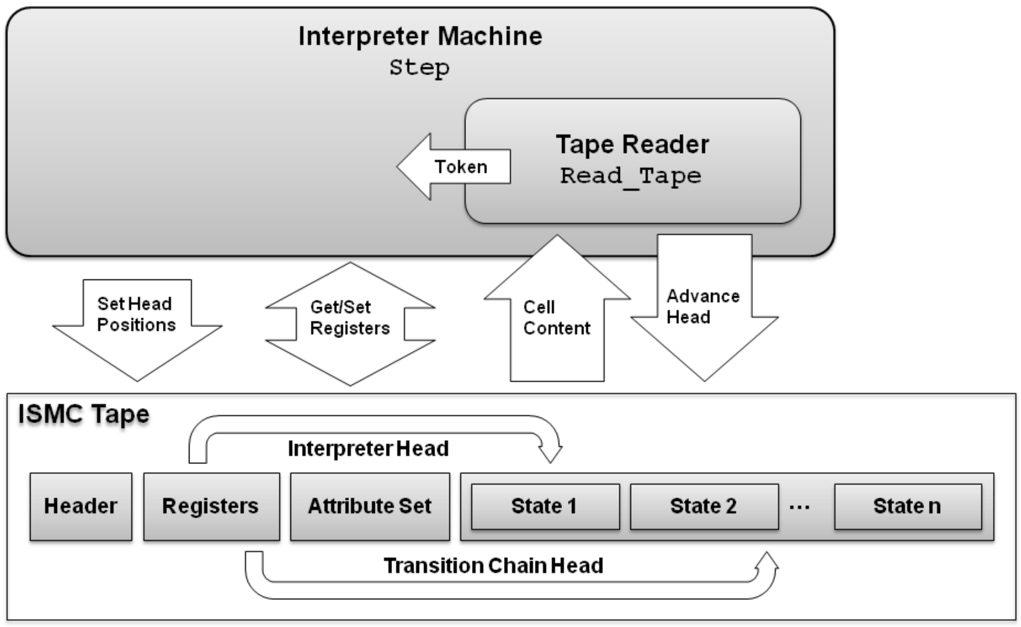


Figure 4.8: Interpretable State Machine Code Overview

interpreter machine. On reading the cell content, the tape reader advances the respective head to the position of the first cell of the subsequent token. The interpreter additionally may execute jumps, i.e. directly setting the heads to specified tape indices, and write to registers where indicated.

formal ISMC

The listing 4.10 defines the ISMC language syntax in Extended Backus-Naur Form (EBNF)[15].

Listing 4.10: ISMC Syntax in EBNF

```
Interpretable State Machine Code = Header ,
                                   Registers ,
                                   Attribute Set ,
                                   States ;

Header = Name , State Machine ID , Initial State Reference ;
Name = 10 * Byte ;
State Machine ID = 4 * Byte ;
Initial State Reference = ISMC Index ;

Registers = Interpreter Head ,
            Index Marker ,
            External Operation Marker ,
            Status ,
            Transition Source ,
```

[15]Intuitively, ASN.1 would have been a suitable candidate, especially with PER. However, it would introduce unwanted overhead into the binary format, similarly TLV encoding, contradicting our optimized approach that is content with TV.

```
              Transition Chain Head ,
              Current Level ,
              Active State Configuration ,
              Exit Action Markers ,
              Assignment Marker ,
              Last Message Buffer ,
              Temporary Message Buffer ,
              Transition Trigger ,
              Integer Register ,
              Interpreter State ;

Interpreter Head = ISMC Index ;
Index Marker = { ISMC Index }- ;
External Operation Marker = ISMC Index ;
Status = GUARD TRUE | GUARD FALSE ;
Transition Source = State ID ;
Transition Chain Head = ISMC Index ;
Current Level = Hierarchy Marker ;
Active State Configuration = { State ID }- ;
Exit Action Marker = { ISMC Index }- ;
Assignment Marker = ISMC Index ;
Last Message Buffer = Message ;
Temporary Message Buffer = Message ;
Transition Trigger = Message ;
Integer Register = Integer ;
Interpreter State = State ID ;

Attribute Set = Variable Count , Variables ;
Variable Count = Integer ;
Variables = { Variable } ;
Variable = n * Byte ; (* Application-specific *)

(* The ISMC Interpreter Instruction Set *)
States = { State }- ;

State = Simple State | Composite State ;

Simple State = SIMPLE STATE , State ID ,
               [ Completion ] ,
               [ Entry Actions ] ,
               [ Exit Actions ] ,
               [ Transitions ] ;

Composite State = COMPOSITE STATE , State ID ,
                  [ Entry Actions ] ,
                  [ CS Exit Actions ] ,
                  EOL ,
                  [ Transitions ] ;

Completion = STOP ;
Entry Actions = ENTRY ACTIONS , { Action }- ;
Exit Actions = EXIT ACTIONS , Length , { Action }- , EOL ;
CS Exit Actions = EXIT ACTIONS , { Action }- ;

Transitions = TRANSITIONS , { Transition }- ;
Transition = TRANSITION ,
             [ Event ] , [ Guard ] ,
             Transition Chain Marker ,
             Source ,
             Next Transition Reference ,
             [ Transition Actions ] ,
             [ Targets ] ;
```

```

Event = EVENT , Message ;
Message = Message ID , Parameter Count , { Parameter } ;
Guard = GUARD , Specific Guard ;
Specific Guard = ( NATIVE_GUARD , Native Guard )
               | ( EXTERNAL_GUARD , External Guard ) ;
Native Guard = ISMC Index , Integer ;
External Guard = Guard ID , Length , Parameter Count , { Parameter } ;
Transition Chain Marker = TC MARKER , ISMC Index ;
Source = State ID ;
Next Transition Reference = ISMC Index ;
Transition Actions = TRANSITION ACTIONS , { Action }- ;
Targets = TARGETS , { Target }- , EOL ;
Target = State Reference , [ Inbound Flag ] ;
Inbound Flag = INBOUND | OUTBOUND      ;
State Reference = ISMC Index ;

State ID = Byte ;
Action = ( NATIVE ACTION , Native Action )
       | ( EXTERNAL ACTION , External Action ) ;
Native Action = Send Message | Assign Value ;
Send Message = SEND , Message ID , { Parameter } ;
Message ID = Byte ;
Assign Value = ASSIGN , Action Variable , Constant ;
External Action = Action ID , Length , Parameter Count , { Parameter } ;
Action ID = 2 * Byte ;
Parameter = Action Variable | Constant ;
Action Variable = ISMC Index ;
Constant = Type , Length , Value ;

Type = Byte ;
Length = ISMC Index ;
ISMC Index = 2 * Byte ;
Integer = 4 * Byte ;

(* All capital-letter definitions are resolved to application-specific
byte terminals. *)
```

header The ISMC *Header* consists of an informal *Name*, a unique *ID*, which we propose could serve as fingerprint if computed as some hash (e.g. current issues of *Message Digest* or *Secure Hash Algorithm*, cmp. [Sc96]; sequence length needs alignment) over the *States*. The *Initial State Reference* indicates the starting cell of the ISMC program in the *States* sequence, similar to a start marker in assembler.

registers *Registers* provide direct access storage for information, i.e. they offer unique addresses respectively offsets. It is not necessary to position a head to read from or write to a register.

Interpreter Head is the primary read/write head of the interpreter machine; the quasi program counter.

Index Marker is used to temporarily store return addresses, e.g. to reiterate a transition sequence.

External Operation Marker indicates the starting cell of an operation invocation specification for a call by the composing component.

Status contains the result of a guard condition evaluation.

Transition Source stores the source state of a current transition for checks regarding high-level transitions.

Transition Chain Head is the secondary head used to iterate the states visited by direct transitions and set the primary head accordingly on each of these states.

Current Level indicates the depth of hierarchical composite state aggregation for the active state.

Active State Configuration stores the currently active state on each hierarchy level.

Exit Action Markers indicate the starting cells of exit actions of each composite state for the current depth.

Assignment Marker temporarily stores the cell index of a variable to assign a value to.

Last Message Buffer contains the message last received by the component's input queue.

Temporary Message Buffer stores messages to send via the component's output queue.

Transition Trigger temporarily stores the message that triggered a transition for later event consumption reference.

Interpreter State contains the current state of the interpreter machine.

application attributes

The *Attribute Set* provides the storage for all application specific variables. Generally, the HIRTE component containing the interpreter declares a corresponding record that maps its address to the attribute set cell sequence in the ISMC. Thus, the binary representation format is application-specific and not given in the syntax definition. Merely a *Variable Count* is proposed and a generic byte sequence reserved.

body of states as instructions

The *States* specification effectively defines the ISMC interpreter instruction set, as each element prompts the interpreter to transition and process the actions defined for its current state, i.e. execute the program given by the ISMC. The state automaton of figures 4.9 to 4.12 exhaustively defines the operational semantics of the *States* productions – as explained in 3.3.1, the semantics in this case relate to and depend on the actual run-time machine that is realized by the interpreter. Each byte opcode changes the state of the RTE in the way specified by the ISMC interpreter automaton.

recalling the SRSM pattern

The underlying modus operandi of composite state processing, direct and high-level transitions corresponds to the mechanisms established in 4.2.1. The same holds true for the supported statechart elements: simple states with entry and exit actions, single region composite states with entry and exit actions, initial and final pseudostates, direct, high-level, explicit and completion transitions with actions, triggered by events respectively message reception and/or guards.

a plain and flat automaton

The interpreter automaton was consciously structurally specified as simple (and thus robust) as possible without composite states or transition actions. This aims at an intelligible run-time model and execution semantics: e.g. a transition only affects the interpreter state, it does not alter the state of the application or the ISMC, respectively. Implementation and also extension are therefore mechanistic and straightforward, minimizing ambiguity.

creating ISMC programs

Fig. 4.13 sketches the business process and artifacts intended for the generation and utilization of ISMC. This work provides the crucial *HIRTE conventions and constraints* for modeling (sections 3.2, 3.3), *formal ISMC language definition* (listing 4.10) and *ISMC interpreter automaton* (figures 4.9 to 4.12, listings 4.11 and 4.12). Other steps of the process have to be based on these results, but are out of scope of this work. For the creation and modification of the model, a common UML tool like Enterprise Architect ([SS10]) is suitable, able to export in XML respectively XMI format. With the language defined, compilation from XML to ISMC is a standard exercise. A rewarding subject for future extensions should lie in the integration of model checking techniques into the process: e.g. a transformation of the model into process algebra terms respectively systems specification languages (CSP [Ro98], [Ho04], PROMELA [Ho03]) allows for the verification and successive ensuring of semantical properties like consistency (cmp. [Ste032]), schedulability and collision-free concurrency.

4.2.3.3 Structure

The structure of the VCU pattern is described by fig. 4.14.

4.2.3.4 Collaboration

- *HIRTE Component*
  Represents a service respectively an application element with restricted interfaces, its activity specification conforming to a state machine structure. Unlike the HIRTE Component of 4.2.1, this one is not necessarily an active task itself, i.e. it is not required to implement and run it as a thread. Instead, either this (passive) component's *Run* or *Step* method is invoked in the context of another active execution unit, which we discuss in the next chapter 5.

  Interactions with other components read from or write to the *Input* and *Output Queue*s. The record *App Variables* composes all attributes the

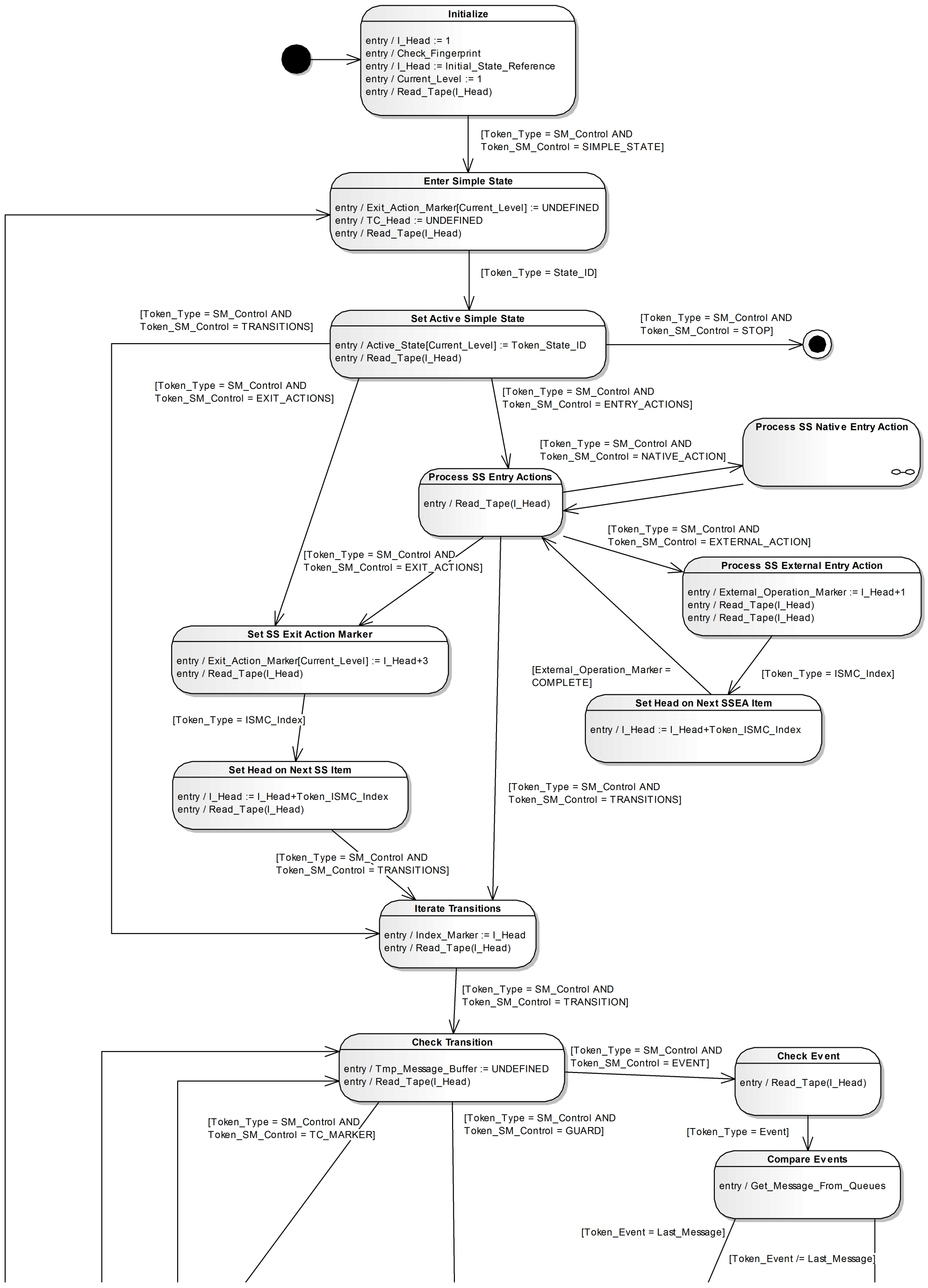

Initialize
entry / I_Head := 1
entry / Check_Fingerprint
entry / I_Head := Initial_State_Reference
entry / Current_Level := 1
entry / Read_Tape(I_Head)
[Token_Type = SM_Control AND Token_SM_Control = SIMPLE_STATE]
Enter Simple State
entry / Exit_Action_Marker[Current_Level] := UNDEFINED
entry / TC_Head := UNDEFINED
entry / Read_Tape(I_Head)
[Token_Type = State_ID]
[Token_Type = SM_Control AND Token_SM_Control = TRANSITIONS]
Set Active Simple State
entry / Active_State[Current_Level] := Token_State_ID
entry / Read_Tape(I_Head)
[Token_Type = SM_Control AND Token_SM_Control = STOP]
[Token_Type = SM_Control AND Token_SM_Control = EXIT_ACTIONS]
[Token_Type = SM_Control AND Token_SM_Control = ENTRY_ACTIONS]
Process SS Native Entry Action
[Token_Type = SM_Control AND Token_SM_Control = NATIVE_ACTION]
Process SS Entry Actions
entry / Read_Tape(I_Head)
[Token_Type = SM_Control AND Token_SM_Control = EXTERNAL_ACTION]
[Token_Type = SM_Control AND Token_SM_Control = EXIT_ACTIONS]
Process SS External Entry Action
entry / External_Operation_Marker := I_Head+1
entry / Read_Tape(I_Head)
entry / Read_Tape(I_Head)
Set SS Exit Action Marker
entry / Exit_Action_Marker[Current_Level] := I_Head+3
entry / Read_Tape(I_Head)
[External_Operation_Marker = COMPLETE]
[Token_Type = ISMC_Index]
Set Head on Next SSEA Item
entry / I_Head := I_Head+Token_ISMC_Index
[Token_Type = ISMC_Index]
Set Head on Next SS Item
entry / I_Head := I_Head+Token_ISMC_Index
entry / Read_Tape(I_Head)
[Token_Type = SM_Control AND Token_SM_Control = TRANSITIONS]
[Token_Type = SM_Control AND Token_SM_Control = TRANSITIONS]
Iterate Transitions
entry / Index_Marker := I_Head
entry / Read_Tape(I_Head)
[Token_Type = SM_Control AND Token_SM_Control = TRANSITION]
Check Transition
entry / Tmp_Message_Buffer := UNDEFINED
entry / Read_Tape(I_Head)
[Token_Type = SM_Control AND Token_SM_Control = EVENT]
Check Event
entry / Read_Tape(I_Head)
[Token_Type = SM_Control AND Token_SM_Control = TC_MARKER]
[Token_Type = SM_Control AND Token_SM_Control = GUARD]
[Token_Type = Event]
Compare Events
entry / Get_Message_From_Queues
[Token_Event = Last_Message]
[Token_Event /= Last_Message]

Figure 4.9: ISMC Interpreter Automaton

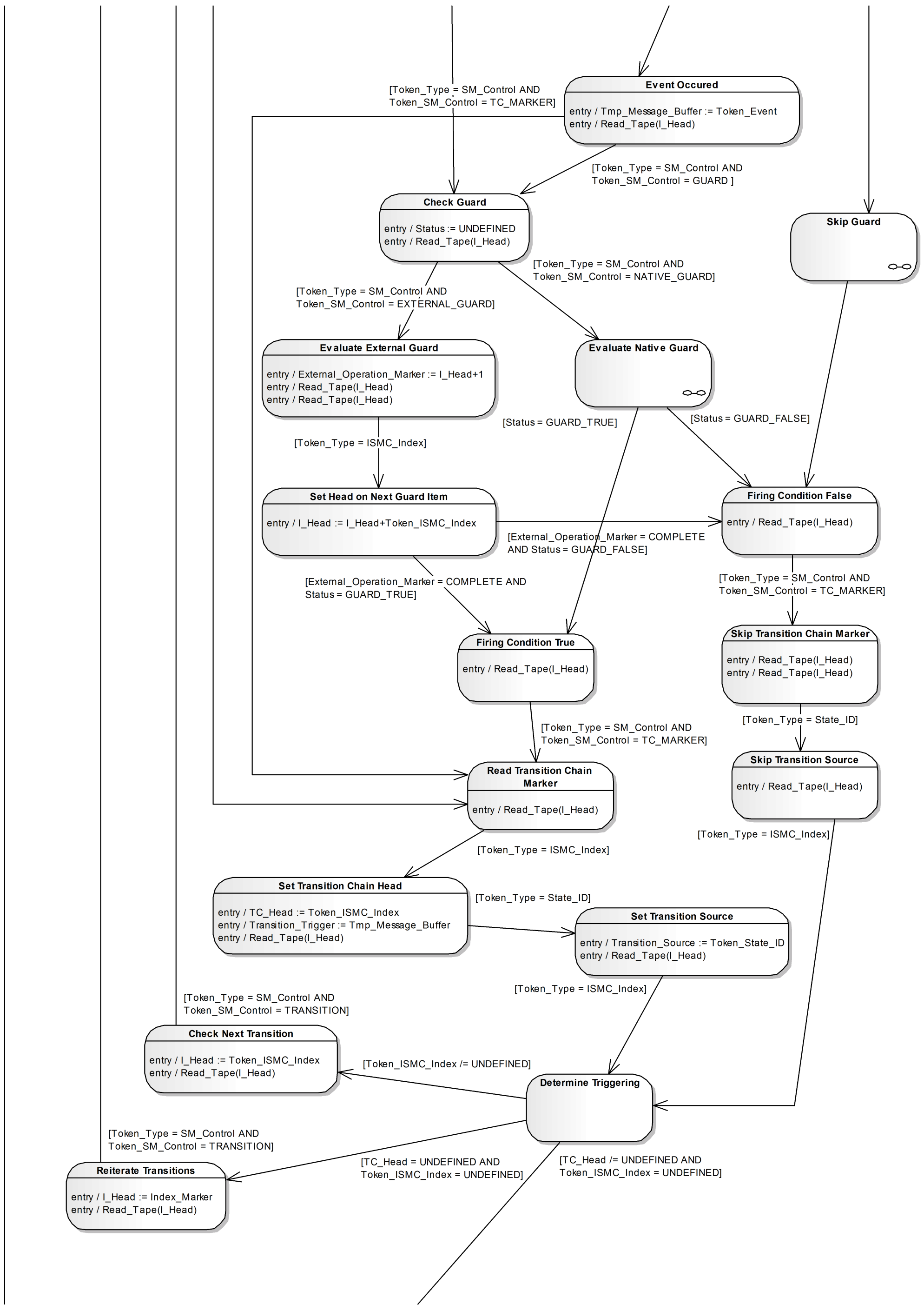

Event Occured
entry / Tmp_Message_Buffer := Token_Event
entry / Read_Tape(I_Head)
[Token_Type = SM_Control AND Token_SM_Control = TC_MARKER]
[Token_Type = SM_Control AND Token_SM_Control = GUARD]
Check Guard
entry / Status := UNDEFINED
entry / Read_Tape(I_Head)
Skip Guard
[Token_Type = SM_Control AND Token_SM_Control = NATIVE_GUARD]
[Token_Type = SM_Control AND Token_SM_Control = EXTERNAL_GUARD]
Evaluate External Guard
entry / External_Operation_Marker := I_Head+1
entry / Read_Tape(I_Head)
entry / Read_Tape(I_Head)
Evaluate Native Guard
[Status = GUARD_TRUE]
[Status = GUARD_FALSE]
[Token_Type = ISMC_Index]
Set Head on Next Guard Item
entry / I_Head := I_Head+Token_ISMC_Index
Firing Condition False
entry / Read_Tape(I_Head)
[External_Operation_Marker = COMPLETE AND Status = GUARD_FALSE]
[External_Operation_Marker = COMPLETE AND Status = GUARD_TRUE]
[Token_Type = SM_Control AND Token_SM_Control = TC_MARKER]
Firing Condition True
entry / Read_Tape(I_Head)
Skip Transition Chain Marker
entry / Read_Tape(I_Head)
entry / Read_Tape(I_Head)
[Token_Type = State_ID]
[Token_Type = SM_Control AND Token_SM_Control = TC_MARKER]
Read Transition Chain Marker
entry / Read_Tape(I_Head)
Skip Transition Source
entry / Read_Tape(I_Head)
[Token_Type = ISMC_Index]
[Token_Type = ISMC_Index]
Set Transition Chain Head
entry / TC_Head := Token_ISMC_Index
entry / Transition_Trigger := Tmp_Message_Buffer
entry / Read_Tape(I_Head)
[Token_Type = State_ID]
Set Transition Source
entry / Transition_Source := Token_State_ID
entry / Read_Tape(I_Head)
[Token_Type = SM_Control AND Token_SM_Control = TRANSITION]
[Token_Type = ISMC_Index]
Check Next Transition
entry / I_Head := Token_ISMC_Index
entry / Read_Tape(I_Head)
[Token_ISMC_Index /= UNDEFINED]
Determine Triggering
[Token_Type = SM_Control AND Token_SM_Control = TRANSITION]
Reiterate Transitions
entry / I_Head := Index_Marker
entry / Read_Tape(I_Head)
[TC_Head = UNDEFINED AND Token_ISMC_Index = UNDEFINED]
[TC_Head /= UNDEFINED AND Token_ISMC_Index = UNDEFINED]

Figure 4.10: ISMC Interpreter Automaton (continued)

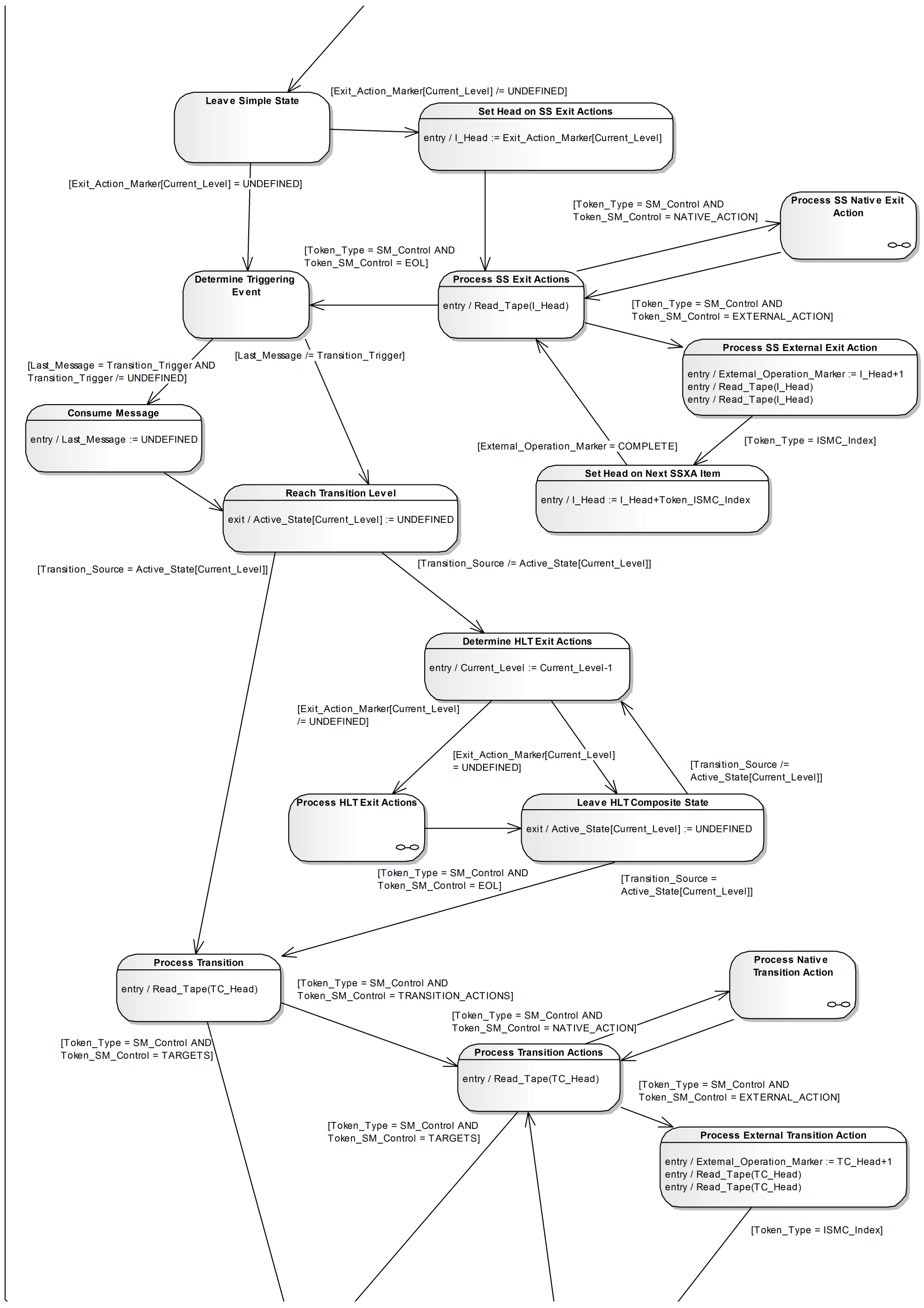

Leave Simple State
[Exit_Action_Marker[Current_Level] /= UNDEFINED]
Set Head on SS Exit Actions
entry / I_Head := Exit_Action_Marker[Current_Level]
[Exit_Action_Marker[Current_Level] = UNDEFINED]
[Token_Type = SM_Control AND Token_SM_Control = NATIVE_ACTION]
Process SS Native Exit Action
[Token_Type = SM_Control AND Token_SM_Control = EOL]
Determine Triggering Event
Process SS Exit Actions
entry / Read_Tape(I_Head)
[Token_Type = SM_Control AND Token_SM_Control = EXTERNAL_ACTION]
Process SS External Exit Action
entry / External_Operation_Marker := I_Head+1
entry / Read_Tape(I_Head)
entry / Read_Tape(I_Head)
[Last_Message = Transition_Trigger AND Transition_Trigger /= UNDEFINED]
[Last_Message /= Transition_Trigger]
Consume Message
entry / Last_Message := UNDEFINED
[External_Operation_Marker = COMPLETE]
[Token_Type = ISMC_Index]
Set Head on Next SSXA Item
entry / I_Head := I_Head+Token_ISMC_Index
Reach Transition Level
exit / Active_State[Current_Level] := UNDEFINED
[Transition_Source = Active_State[Current_Level]]
[Transition_Source /= Active_State[Current_Level]]
Determine HLT Exit Actions
entry / Current_Level := Current_Level-1
[Exit_Action_Marker[Current_Level] /= UNDEFINED]
[Exit_Action_Marker[Current_Level] = UNDEFINED]
[Transition_Source /= Active_State[Current_Level]]
Process HLT Exit Actions
Leave HLT Composite State
exit / Active_State[Current_Level] := UNDEFINED
[Token_Type = SM_Control AND Token_SM_Control = EOL]
[Transition_Source = Active_State[Current_Level]]
Process Transition
entry / Read_Tape(TC_Head)
[Token_Type = SM_Control AND Token_SM_Control = TRANSITION_ACTIONS]
Process Native Transition Action
[Token_Type = SM_Control AND Token_SM_Control = NATIVE_ACTION]
[Token_Type = SM_Control AND Token_SM_Control = TARGETS]
Process Transition Actions
entry / Read_Tape(TC_Head)
[Token_Type = SM_Control AND Token_SM_Control = EXTERNAL_ACTION]
[Token_Type = SM_Control AND Token_SM_Control = TARGETS]
Process External Transition Action
entry / External_Operation_Marker := TC_Head+1
entry / Read_Tape(TC_Head)
entry / Read_Tape(TC_Head)
[Token_Type = ISMC_Index]

Figure 4.11: ISMC Interpreter Automaton (continued)

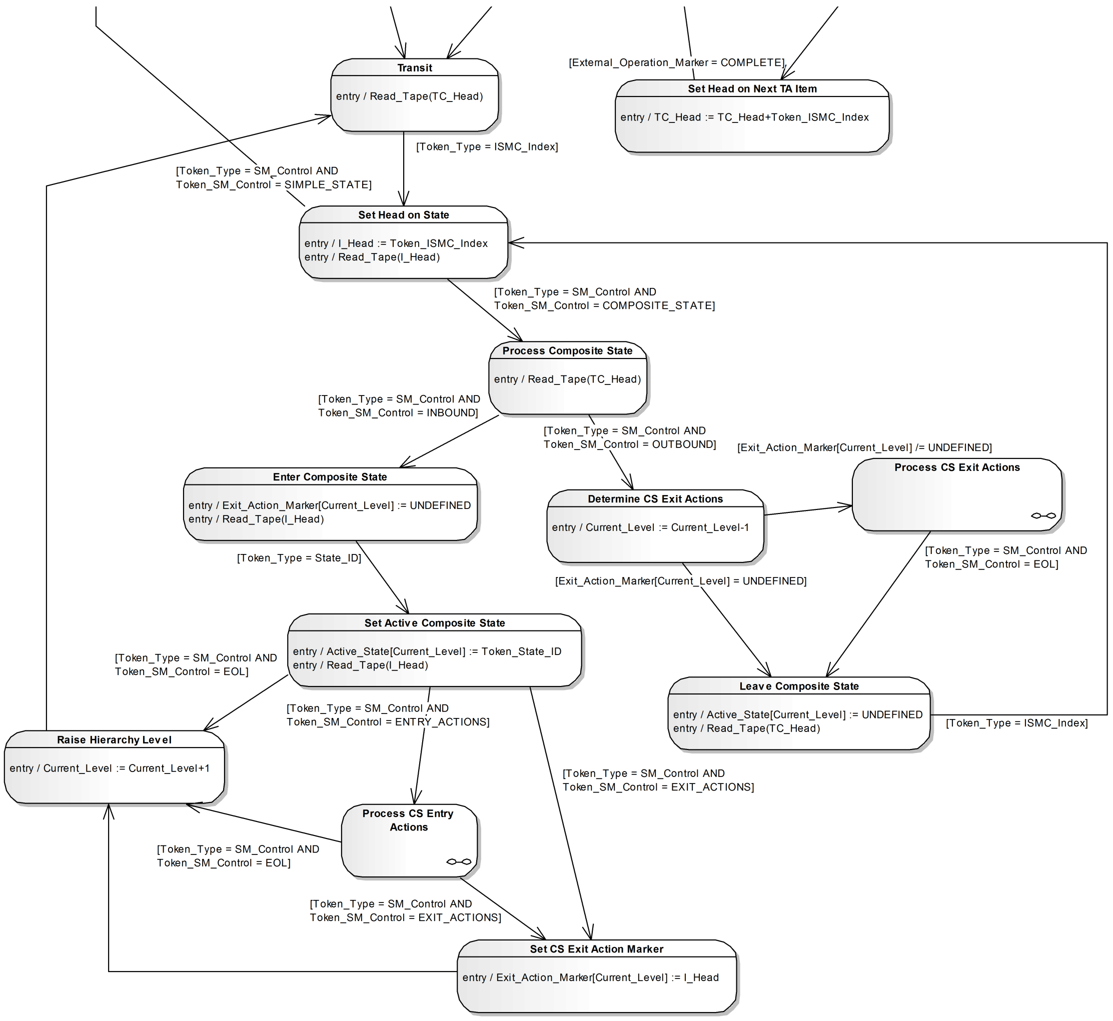

Transit
entry / Read_Tape(TC_Head)
[External_Operation_Marker = COMPLETE]
Set Head on Next TA Item
entry / TC_Head := TC_Head+Token_ISMC_Index
[Token_Type = ISMC_Index]
[Token_Type = SM_Control AND Token_SM_Control = SIMPLE_STATE]
Set Head on State
entry / I_Head := Token_ISMC_Index
entry / Read_Tape(I_Head)
[Token_Type = SM_Control AND Token_SM_Control = COMPOSITE_STATE]
Process Composite State
entry / Read_Tape(TC_Head)
[Token_Type = SM_Control AND Token_SM_Control = INBOUND]
[Token_Type = SM_Control AND Token_SM_Control = OUTBOUND]
[Exit_Action_Marker[Current_Level] /= UNDEFINED]
Process CS Exit Actions
Enter Composite State
entry / Exit_Action_Marker[Current_Level] := UNDEFINED
entry / Read_Tape(I_Head)
Determine CS Exit Actions
entry / Current_Level := Current_Level-1
[Token_Type = State_ID]
[Exit_Action_Marker[Current_Level] = UNDEFINED]
[Token_Type = SM_Control AND Token_SM_Control = EOL]
Set Active Composite State
entry / Active_State[Current_Level] := Token_State_ID
entry / Read_Tape(I_Head)
[Token_Type = SM_Control AND Token_SM_Control = EOL]
Leave Composite State
entry / Active_State[Current_Level] := UNDEFINED
entry / Read_Tape(TC_Head)
[Token_Type = ISMC_Index]
[Token_Type = SM_Control AND Token_SM_Control = ENTRY_ACTIONS]
Raise Hierarchy Level
entry / Current_Level := Current_Level+1
[Token_Type = SM_Control AND Token_SM_Control = EXIT_ACTIONS]
Process CS Entry Actions
[Token_Type = SM_Control AND Token_SM_Control = EOL]
[Token_Type = SM_Control AND Token_SM_Control = EXIT_ACTIONS]
Set CS Exit Action Marker
entry / Exit_Action_Marker[Current_Level] := I_Head

Figure 4.12: ISMC Interpreter Automaton (continued, finished)

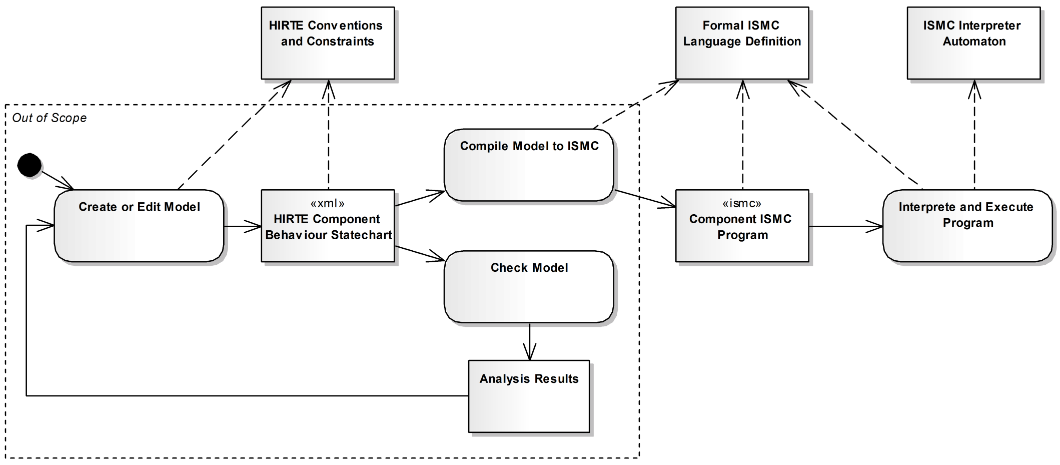

HIRTE Conventions and Constraints
Formal ISMC Language Definition
ISMC Interpreter Automaton
Out of Scope
Compile Model to ISMC
Create or Edit Model
«xml» HIRTE Component Behaviour Statechart
«ismc» Component ISMC Program
Interprete and Execute Program
Check Model
Analysis Results

Figure 4.13: ISMC Business Process

component application requires for processing, same as *App Ops* contains all procedures referenced by actions or guards of the application's state machine.

The component is able to execute *n* ISMC programs by managing *n* composed ISMC Interpreters: it receives a program on calls of *Set Tape* and passes it to the corresponding *Machine*. After that, *Run* prompts the component to invoke the *Machine*s in a continuous loop, *Step* advances processing one state of each interpreter machine (not of the service realized by the component).

The existence of a superordinate control instance notwithstanding, the *Run* method may already apply some local scheduling strategy (cmp. [BW09], chapter 11) to the invocation of a sequence of interpreters by considering the measurable *number of interpreter transitions*, the *number of application transitions*, the *number of actions and guards* (external operations), and *waiting for events/messages*. The frequency of calls to *Step* of each *Machine* can then be adjusted to given ISMC program *priorities*, whether the realized activity is *sporadic or aperiodic* and its operation *run-time requirements* (as an approximation of temporal requirements). Correspondingly, *Step* is meant for scheduling by an instance controlling a composition of VCUs (ref. to chapter 5).

The separation of application and generic state machine concerns implies a direct handle of the component to the ISMC byte sequence. This pertains to the execution of external operations (see below); a mechanism that is not entirely encapsuled by the *Get* and *Set* methods of the interpreter.

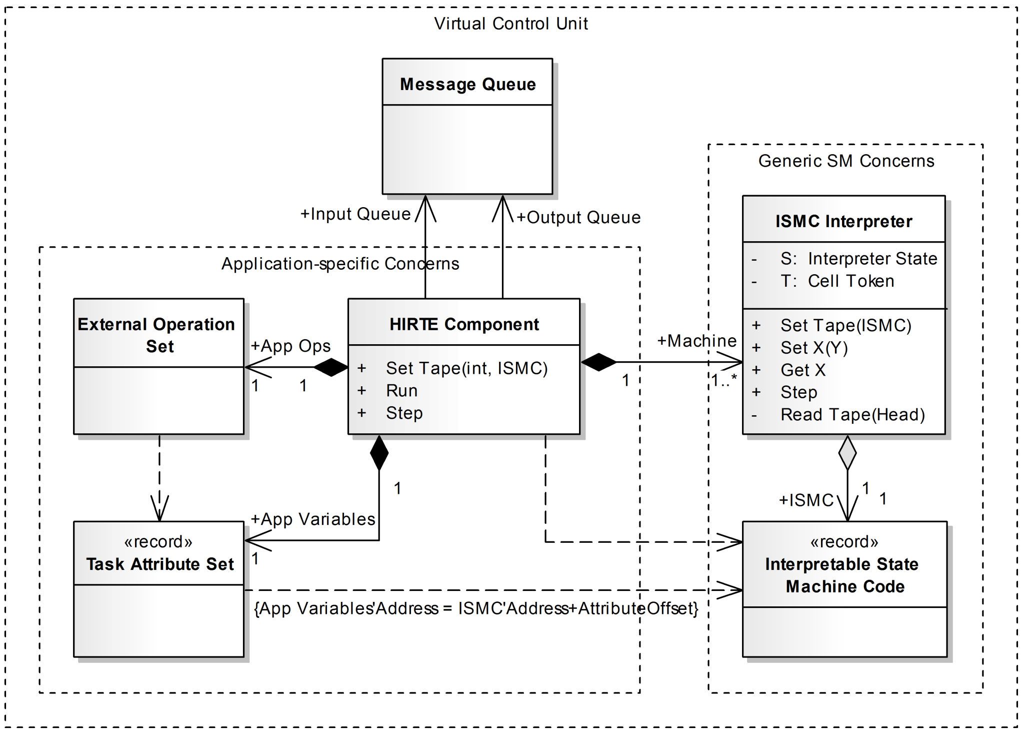


Figure 4.14: Virtual Control Unit Pattern Structure

- *Task Attribute Set*
  Composes all standard type, array and structure variables of the application realized by the VCU. All operations, i.e. actions and guards, work with this set.

  A critical additional property is the relation to the ISMC byte sequence: the addresses of the variable declarations map to a subsequence reserved for the attribute set. This works toward achieving fulfillment of requirement VII (persistency of activities) by containing all elements relevant to the system state as a precondition for structured storage.

- *External Operation Set*
  The ISMC interpreter differentiates between *native* and *external* operations. Generally, these encompass *actions* as well as *guards*, i.e. procedures that check guard conditions. While native operations are part of the ISMC instruction set and are thus processed by the interpreter, external operations call for separate execution.

  To this end, an *external operation marker* registers an address in the ISMC byte array that tags the specification (ID, parameters) of the operation to invoke. After each *Step* of the interpreter, the HIRTE component checks the register with *Get_ External_ Operation_ Marker*. If the value is not equal to *UNDEFINED*, the component directly accesses the ISMC at the given index, retrieving the operation's ID and, if given, parameters.

Conforming to a mapping of the IDs to the actions and guards of the set, the corresponding operation is invoked. As stated above, it may access variables of the *Task Attribute Set* and additionally the ISMC registers, so a guard may *Set_ Status* to *GUARD_ TRUE* or *GUARD_ FALSE* after evaluation. Thus, on return, the component merely notifies the interpreter with *Set_ External_ Operation_ Marker*(*COMPLETE*).

- *Message Queue*
  The input and output queues for interaction with other components have the same properties as given in 4.2.1.3. To achieve the handover of messages between queue interface structures and ISMC, the component utilizes ISMC registers with the *Get*/*Set* methods of the interpreter.

  If an action prompts the ISMC program to send a message, it writes it to the *Temporary Message Buffer* register. In the loop of *Step*s, the component checks the register for messages (*Get_ Tmp_ Message_ Buffer* not equal *UNDEFINED_ MSG*). On a stored message, the component writes it to the output queue and sets the register to *UNDEFINED_ MSG*.

  Whenever the component reads a message from its input queue in the interpreter processing loop, it writes it to the ISMC tape with *Set_ - Last_ Message_ Buffer*, if the previous message was already consumed by the ISMC program (register equals *UNDEFINED_ MSG*), i.e. consumption is handled by an interpreter state.

- *ISMC Interpreter*
  Implements the ISMC interpreter automaton of the previous section. All attributes necessary for ISMC processing are mapped to the *ISMC* tape, so the overall system state is contained in this structure, working toward fulfillment of requirement VII (persistency of activities). The possible exception are the variables of *Cell Token*s and types used by the tape reader procedure. However, this is uncritical, as the current and last read tokens may be reproduced from the position of the *Interpreter Head* (see above), which in turn is safely stored in an ISMC register.

  With *Set Tape*, the HIRTE Component passes a handle to the ISMC to process. *Get* and *Set* methods provide access to the registers and other ISMC attributes, where applicable. *Step* invokes the interpreter processing and the transition of the automaton states.

  Actions of the interpreter states rely on *Read Tape* for compiling cell content at the position of a given head to tokens. This implies the conversion of the byte sequence to tokens of

SM Control – an instruction affecting the machine's control flow,

State ID – the identification of an application state,

ISMC Index – a position on the ISMC tape,

Byte – some byte value,

Double – two bytes,

Integer – some integer value and

Event – a message byte sequence with ID and content respectively message body.

The token type is explicitly provided in an interpreter attribute for discrimination after each call to *Read Tape.*

Note: another convenient approach might have been to implement the tape reader as a class and associate the interpreter with an instance. This consequent encapsulation would support replacement of the byte scanner in cases of other potential ISMC formats without needing to change the interpreter itself. However, we banned dispatching in section 3.3.2: the call to *Tape Reader.Read Tape* would have to be resolved dynamically. Thus, the alternative scanner procedure still provides us with a degree of encapsulation (procedure instead of class, defined set of associated attributes) and clear boundaries for future adaptions.

- *Interpretable State Machine Code*
  A byte array structurally conforming to the specification of listing 4.10 (ISMC Syntax in EBNF). It contains the HIRTE component's application program, its variables and values, and the current state of execution.

application and state machine concerns

The given collaboration achieves a separation of concerns of the application and the generic state machine mechanisms. The component realizes the application per se, the attribute and operation sets are directly associated with the application's domain, providing its corresponding data and specialized functionality if needed. These application specifics take a generic form in the context of the interpreter and ISMC: native operations will usually be of general utility, application states are not interpreted beyond their byte value and the application attributes map to an opaque byte sequence.

vcu stereotype

In the following, we will interpret the VCU boundary as a component; a stereotype «*vcu* »will indicate that it contains the setup as given above.

4.2.3.5 Implementation

The following listings provide a reference implementation for the elements of a Virtual Control Unit. To keep the extensiveness of the sources in check, basic modules (message queues, operation sets) as well as various checks have been omitted; like an initial signature check of the ISMC body's hash and especially concerning the explicit conversions of binary formats. This does not affect the essence of the solution.

Listing 4.11: ISMC Interpreter Spec

```
with System.Storage_Elements,
     Ada.Storage_IO,
     Ada.Integer_Text_IO,
     Ada.Unchecked_Conversion,
     HIRTE_Configuration,
     HIRTE_Communications,
     Ada.Text_IO;

use System.Storage_Elements,
    Ada.Integer_Text_IO,
    HIRTE_Configuration,
    HIRTE_Communications,
    Ada.Text_IO;

package HIRTE_ISMC_Interpreter is

   -- Configuration and Register Offsets
   ISMC_LENGTH : constant Storage_Count := 5000;
   MAX_HIERARCHY_DEPTH : constant Storage_Count := 3;

   ID_OFFSET : constant Storage_Count := 11;
   INITIAL_STATE_REF_OFFSET : constant Storage_Count := 15;
   I_HEAD_OFFSET : constant Storage_Count := 17;
   INDEX_MARKER_OFFSET : constant Storage_Count := 19;
   EXTERNAL_OPERATION_MARKER_OFFSET : constant Storage_Count := 23;
   STATUS_OFFSET : constant Storage_Count := 25;
   TRANSITION_SOURCE_OFFSET : constant Storage_Count := 26;
   TC_HEAD_OFFSET : constant Storage_Count := 27;
   CURRENT_LEVEL_OFFSET : constant Storage_Count := 29;
   ACTIVE_STATE_OFFSET : constant Storage_Count := 30;
   EXIT_ACTION_MARKERS_OFFSET : constant Storage_Count := 34;
   ASSIGNMENT_MARKER_OFFSET : constant Storage_Count := 42;
   LAST_MESSAGE_BUFFER_OFFSET : constant Storage_Count := 44;
   TMP_MESSAGE_BUFFER_OFFSET : constant Storage_Count := 54;
   TRANSITION_TRIGGER_OFFSET : constant Storage_Count := 64;
   INTEGER_REGISTER_OFFSET : constant Storage_Count := 74;

   INTERPRETER_STATE_OFFSET : constant Storage_Count := 92;
   HEADER_OFFSET : constant Storage_Count := 93;

   -- Types and Constants
   subtype ISMC_Tape is Storage_Array(1..ISMC_LENGTH);
   type ISMC_Tape_Ref is access all ISMC_Tape;
   type ISMC_Index is range 0..ISMC_LENGTH;
   for ISMC_Index'Size use 16;
   UNDEFINED : constant ISMC_Index := 0;
   COMPLETE : constant ISMC_Index := 1;
   UNDEFINED_MSG : constant Messages.Binary_Buffer :=
     (16#00#,16#00#,16#00#,16#00#,16#00#,16#00#,16#00#,16#00#,16#00#,16#00#);
   UNDEFINED_STATE : constant Storage_Element := 16#00#;
   subtype ISMC_Index_Cell is Storage_Array(1..2);
   type Hierarchy_Marker is range 0..MAX_HIERARCHY_DEPTH;
   for Hierarchy_Marker'Size use 8;
   subtype Active_State_Configuration is Storage_Array(0..MAX_HIERARCHY_DEPTH);
   subtype Integer_Buffer is Storage_Array(1..4);

   type ISMC_Automaton_State is (Initial, Initialize, Enter_Simple_State,
        Set_Active_Simple_State, Process_SS_Entry_Actions,
        Process_SS_Native_Entry_Action, Process_SS_External_Entry_Action,
        Set_Head_on_Next_SSEA_Item, Set_SS_Exit_Action_Marker,
        Set_Head_on_Next_SS_Item, Iterate_Transitions, Check_Transition,
        Check_Event, Compare_Events, Event_Occured, Check_Guard,
        Evaluate_Native_Guard, Skip_Guard, Evaluate_External_Guard,
        Set_Head_on_Next_Guard_Item, Firing_Condition_False,
        Firing_Condition_True, Skip_Transition_Chain_Marker,
        Read_Transition_Chain_Marker, Set_Transition_Chain_Head,
        Skip_Transition_Source, Check_Next_Transition, Set_Transition_Source,
        Reiterate_Transitions, Determine_Triggering, Set_Head_on_SS_Exit_Actions,
        Process_SS_Exit_Actions, Process_SS_Native_Exit_Action,
        Process_SS_External_Exit_Action, Set_Head_on_Next_SSXA_Item,
        Leave_Simple_State, Determine_Triggering_Event, Consume_Message,
        Reach_Transition_Level, Determine_HLT_Exit_Actions, Process_Transition,
        Set_Head_on_HLT_Exit_Actions, Process_HLT_Exit_Actions,
        Process_HLT_Native_Exit_Action, Process_HLT_External_Exit_Action,
        Set_Head_on_Next_HLTXA_Item, Leave_HLT_Composite_State, Transit,
        Process_Transition_Actions, Process_Native_Transition_Action,
        Process_External_Transition_Action, Set_Head_on_Next_TA_Item,
        Set_Head_on_State, Process_Composite_State, Enter_Composite_State,
        Set_Active_Composite_State, Process_CS_Entry_Actions,
        Process_CS_Native_Entry_Action, Process_CS_External_Entry_Action,
        Set_Head_on_Next_CSEA_Item, Set_CS_Exit_Action_Marker,
        Raise_Hierarchy_Level, Determine_CS_Exit_Actions,
        Set_Head_on_CS_Exit_Actions, Process_CS_Exit_Actions,
        Process_CS_Native_Exit_Action, Process_CS_External_Exit_Action,
        Set_Head_on_Next_CSXA_Item, Leave_Composite_State, Final, Failure);

    for ISMC_Automaton_State use (Initial => 0, Initialize => 1,
        Enter_Simple_State => 2, Set_Active_Simple_State => 3,
        Process_SS_Entry_Actions => 4, Process_SS_Native_Entry_Action => 5,
        Process_SS_External_Entry_Action => 6, Set_Head_on_Next_SSEA_Item => 7,
        Set_SS_Exit_Action_Marker => 8, Set_Head_on_Next_SS_Item => 9,
        Iterate_Transitions => 10, Check_Transition => 11, Check_Event => 12,
        Compare_Events => 13, Event_Occured => 14, Check_Guard => 15,
        Evaluate_Native_Guard => 16, Skip_Guard => 17,
        Evaluate_External_Guard => 18, Set_Head_on_Next_Guard_Item => 19,
        Firing_Condition_False => 20, Firing_Condition_True => 21,
        Skip_Transition_Chain_Marker => 22, Read_Transition_Chain_Marker => 23,
        Set_Transition_Chain_Head => 24, Skip_Transition_Source => 25,
        Check_Next_Transition => 26, Set_Transition_Source => 27,
        Reiterate_Transitions => 28, Determine_Triggering => 29,
        Set_Head_on_SS_Exit_Actions => 30, Process_SS_Exit_Actions => 31,
        Process_SS_Native_Exit_Action => 32, Process_SS_External_Exit_Action => 33,
        Set_Head_on_Next_SSXA_Item => 34, Leave_Simple_State => 35,
        Determine_Triggering_Event => 36, Consume_Message => 37,
        Reach_Transition_Level => 38, Determine_HLT_Exit_Actions => 39,
        Process_Transition => 40, Set_Head_on_HLT_Exit_Actions => 41,
        Process_HLT_Exit_Actions => 42, Process_HLT_Native_Exit_Action => 43,
        Process_HLT_External_Exit_Action => 44, Set_Head_on_Next_HLTXA_Item => 45,
        Leave_HLT_Composite_State => 46, Transit => 47,
        Process_Transition_Actions => 48, Process_Native_Transition_Action => 49,
        Process_External_Transition_Action => 50, Set_Head_on_Next_TA_Item => 51,
        Set_Head_on_State => 52, Process_Composite_State => 53,
        Enter_Composite_State => 54, Set_Active_Composite_State => 55,
        Process_CS_Entry_Actions => 56, Process_CS_Native_Entry_Action => 57,
        Process_CS_External_Entry_Action => 58, Set_Head_on_Next_CSEA_Item => 59,
        Set_CS_Exit_Action_Marker => 60, Raise_Hierarchy_Level => 61,
        Determine_CS_Exit_Actions => 62, Set_Head_on_CS_Exit_Actions => 63,
        Process_CS_Exit_Actions => 64, Process_CS_Native_Exit_Action => 65,
        Process_CS_External_Exit_Action => 66, Set_Head_on_Next_CSXA_Item => 67,
      Leave_Composite_State => 68, Final => 69, Failure => 70);

   for ISMC_Automaton_State'Size use 8;
   for ISMC_Automaton_State'Alignment use 1;

   type ISMC_Token_Type is (T_SM_Control, T_State_ID, T_ISMC_Index, T_Event,
      T_Byte, T_Double, T_Integer);
   for ISMC_Token_Type use (T_SM_Control => 1, T_State_ID => 2, T_ISMC_Index => 3,
      T_Event => 4, T_Byte => 5, T_Double => 6, T_Integer => 7);
   for ISMC_Token_Type'Size use 8;
   type ISMC_Control_Item is (SIMPLE_STATE, STOP, TRANSITIONS, ENTRY_ACTIONS,
      EXIT_ACTIONS, NATIVE_ACTION, EXTERNAL_ACTION, TRANSITION, EVENT, TC_MARKER,
      GUARD, NATIVE_GUARD, EXTERNAL_GUARD, EOL, TARGETS, TRANSITION_ACTIONS,
      COMPOSITE_STATE, INBOUND, OUTBOUND, SEND, ASSIGN);
   for ISMC_Control_Item use (SIMPLE_STATE => 1, STOP => 2, TRANSITIONS => 3,
      ENTRY_ACTIONS => 4, EXIT_ACTIONS => 5, NATIVE_ACTION => 6, EXTERNAL_ACTION => 7,
      TRANSITION => 8, EVENT => 9, TC_MARKER => 10, GUARD => 11, NATIVE_GUARD => 12,
      EXTERNAL_GUARD => 13, EOL => 14, TARGETS => 15, TRANSITION_ACTIONS => 16,
      COMPOSITE_STATE => 17, INBOUND => 18, OUTBOUND => 19, SEND => 20, ASSIGN => 21);
   for ISMC_Control_Item'Size use 8;
   type ISMC_Status is (GUARD_FALSE, GUARD_TRUE, STATUS_UNDEFINED);
   for ISMC_Status use (GUARD_FALSE => 0, GUARD_TRUE => 1, STATUS_UNDEFINED => 2);
   for ISMC_Status'Size use 8;

   protected type ISMC_Automaton is

      procedure Set_Tape(T : in ISMC_Tape_Ref);
      function Get_Tape return ISMC_Tape_Ref;
      procedure Set_Interpreter_State(S : in ISMC_Automaton_State);
      function Get_Interpreter_State return ISMC_Automaton_State;
      procedure Set_ISMC_Name(S : in String);
      function Get_ISMC_Name return String;
      procedure Set_ISMC_ID(ID : in Storage_Array);
      function Get_ISMC_ID return Storage_Array;
      procedure Set_Initial_State_Reference(S : in ISMC_Index_Cell);
      function Get_Initial_State_Reference return ISMC_Index_Cell;
      procedure Set_Interpreter_Head(H : in ISMC_Index_Cell);
      function Get_Interpreter_Head return ISMC_Index_Cell;
      procedure Set_Index_Marker(M : in ISMC_Index_Cell);
      function Get_Index_Marker return ISMC_Index_Cell;
      procedure Set_External_Operation_Marker(I : in ISMC_Index);
      function Get_External_Operation_Marker return ISMC_Index;
      procedure Set_Status(S : in ISMC_Status);
      function Get_Status return ISMC_Status;
      procedure Set_Transition_Source(S : in Storage_Element);
      function Get_Transition_Source return Storage_Element;
      procedure Set_Transition_Chain_Head(H : in ISMC_Index_Cell);
      function Get_Transition_Chain_Head return ISMC_Index_Cell;
      procedure Set_Current_Level(L : in Hierarchy_Marker);
      function Get_Current_Level return Hierarchy_Marker;
      procedure Set_Active_State(S : in Storage_Element; i : in Hierarchy_Marker);
      function Get_Active_State(i : in Hierarchy_Marker) return Storage_Element;
      procedure Set_Exit_Action_Marker(M : in ISMC_Index_Cell;
                                       i : in Hierarchy_Marker);
      function Get_Exit_Action_Marker(i : in Hierarchy_Marker)
         return ISMC_Index_Cell;
      procedure Set_Assignment_Marker(M : in ISMC_Index_Cell);
      function Get_Assignment_Marker return ISMC_Index_Cell;
      procedure Set_Last_Message_Buffer(M : in Messages.Binary_Buffer);
      function Get_Last_Message_Buffer return Messages.Binary_Buffer;
      procedure Set_Tmp_Message_Buffer(M : in Messages.Binary_Buffer);
      function Get_Tmp_Message_Buffer return Messages.Binary_Buffer;
      procedure Set_Transition_Trigger(M : in Messages.Binary_Buffer);
      function Get_Transition_Trigger return Messages.Binary_Buffer;

      procedure Read_Tape(Head : in out ISMC_Index_Cell);
      procedure Write_Index_to_Tape(Head : in out ISMC_Index_Cell;
                                    Index : in ISMC_Index);
      procedure Write_Control_Item_to_Tape(Head : in out ISMC_Index_Cell;
                                           Item : in ISMC_Control_Item);
      procedure Step;

      private

      ISMC : ISMC_Tape_Ref;
      Interpreter_State : ISMC_Automaton_State := Initial;
      Token_Type : ISMC_Token_Type;
      Token_SM_Control : ISMC_Control_Item;
      Token_State_ID : Storage_Element;
      Token_ISMC_Index : ISMC_Index;
      Token_Byte : Storage_Element;
      Token_Double : Storage_Array(1..2);
      Token_Integer : Integer;
      Token_Event : Messages.Binary_Buffer;

   end ISMC_Automaton;

end HIRTE_ISMC_Interpreter;
```

Listing 4.11 specifies the ISMC interpreter. For all types relevant to the system state configuration an explicit representation is defined to enable persistency, i.e. structured and systematic storage of the program in a given run-time state of execution.

Listing 4.12: ISMC Interpreter Body

```
package body HIRTE_ISMC_Interpreter is

   protected body ISMC_Automaton is

      procedure Set_Tape(T : in ISMC_Tape_Ref) is
      begin
         ISMC := T;
      end;

      function Get_Tape return ISMC_Tape_Ref is
      begin
         return ISMC;
      end;

      procedure Set_Interpreter_State(S : in ISMC_Automaton_State) is
      begin
         ISMC(INTERPRETER_STATE_OFFSET) := Write_ISMC_Automaton_State(S);
      end;

      function Get_Interpreter_State return ISMC_Automaton_State is
      begin
         return Read_ISMC_Automaton_State(ISMC(INTERPRETER_STATE_OFFSET));
      end;

      procedure Set_ISMC_Name(S : in String) is
      begin
         ISMC(1..10) := Set_Name(S);
```

```ada
   end;

   function Get_ISMC_Name return String is
   begin
      return Get_Name(ISMC(1..10));
   end;

   procedure Set_ISMC_ID(ID : in Storage_Array) is
   begin
      ISMC(ID_OFFSET..ID_OFFSET+3) := ID;
   end;

   function Get_ISMC_ID return Storage_Array is
   begin
      return ISMC(ID_OFFSET..ID_OFFSET+3);
   end;

   procedure Set_Initial_State_Reference(S : in ISMC_Index_Cell) is
   begin
      ISMC(INITIAL_STATE_REF_OFFSET..INITIAL_STATE_REF_OFFSET+1) := S;
   end;

   function Get_Initial_State_Reference return ISMC_Index_Cell is
   begin
      return ISMC(INITIAL_STATE_REF_OFFSET..INITIAL_STATE_REF_OFFSET+1);
   end;

   procedure Set_Interpreter_Head(H : in ISMC_Index_Cell) is
   begin
      ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1) := H;
   end;

   function Get_Interpreter_Head return ISMC_Index_Cell is
   begin
      return ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1);
   end;
   ...

   procedure Read_Tape(Head : in out ISMC_Index_Cell) is
   -- Precondition: Head references a cell containing a token type.
   -- Advancing the head will not exceed the specified ISMC index range.
   -- Accepts an ISMC cell index representing a machine read/write head.
   -- Reads the first byte from the current position of the head,
   -- interpreting it as the type of token to read. It then advances
   -- the head's position by one to n bytes, according to the type to read.
   -- The token type and token contained by the n ISMC bytes are stored
   -- in the discriminator and corresponding attribute, respectively.
   -- Postcondition: Token_Type specifies the read token, Token_X
   -- contains the token, Head references the successor token type.
   begin

      Token_Type :=
         Read_ISMC_Token_Type(ISMC(Storage_Offset(Read_ISMC_Index(Head))));

      case Token_Type is

         when T_SM_Control =>
            Head := Write_ISMC_Index(Read_ISMC_Index(Head)+1);
            Token_SM_Control :=
               Read_ISMC_Control_Item(ISMC(Storage_Offset(Read_ISMC_Index(Head))));
            Head := Write_ISMC_Index(Read_ISMC_Index(Head)+1);

            when T_State_ID =>
                Head := Write_ISMC_Index(Read_ISMC_Index(Head)+1);
                Token_State_ID := ISMC(Storage_Offset(Read_ISMC_Index(Head)));
                Head := Write_ISMC_Index(Read_ISMC_Index(Head)+1);

            when T_ISMC_Index =>
                Head := Write_ISMC_Index(Read_ISMC_Index(Head)+1);
                Token_ISMC_Index :=
                    Read_ISMC_Index(ISMC(Storage_Offset(Read_ISMC_Index(Head))
                        ..Storage_Offset(Read_ISMC_Index(Head))+1));
                Head := Write_ISMC_Index(Read_ISMC_Index(Head)+2);

            when T_Event =>
                Head := Write_ISMC_Index(Read_ISMC_Index(Head)+1);
                Token_Event :=
                    ISMC(Storage_Offset(Read_ISMC_Index(Head))
                        ..Storage_Offset(Read_ISMC_Index(Head))
                           +HIRTE_Configuration.MSG_BUFFER_SIZE_BYTES-1);
                Head := Write_ISMC_Index(Read_ISMC_Index(Head)
                    +ISMC_Index(HIRTE_Configuration.MSG_BUFFER_SIZE_BYTES));

            when T_Byte =>
                Head := Write_ISMC_Index(Read_ISMC_Index(Head)+1);
                Token_Byte := ISMC(Storage_Offset(Read_ISMC_Index(Head)));
                Head := Write_ISMC_Index(Read_ISMC_Index(Head)+1);

            when T_Double =>
                Head := Write_ISMC_Index(Read_ISMC_Index(Head)+1);
                Token_Double := ISMC(Storage_Offset(Read_ISMC_Index(Head))
                    ..Storage_Offset(Read_ISMC_Index(Head))+1);
                Head := Write_ISMC_Index(Read_ISMC_Index(Head)+2);

            when T_Integer =>
                Head := Write_ISMC_Index(Read_ISMC_Index(Head)+1);
                Token_Integer :=
                    Read_Integer_Buffer(ISMC(Storage_Offset(Read_ISMC_Index(Head))
                        ..Storage_Offset(Read_ISMC_Index(Head))+3));
                Head := Write_ISMC_Index(Read_ISMC_Index(Head)+4);

        end case;

    end;

    procedure Write_Index_to_Tape(Head  : in out ISMC_Index_Cell;
                                  Index : in ISMC_Index) is
    begin
        ISMC(Storage_Offset(Read_ISMC_Index(Head))) :=
            Write_ISMC_Token_Type(T_ISMC_Index);
        ISMC(Storage_Offset(Read_ISMC_Index(Head))+1
            ..Storage_Offset(Read_ISMC_Index(Head))+2) := Write_ISMC_Index(Index);
        Head := Write_ISMC_Index(Read_ISMC_Index(Head)+3);
    end;

    procedure Write_Control_Item_to_Tape(Head : in out ISMC_Index_Cell;
                                         Item : in ISMC_Control_Item) is
    begin
        ISMC(Storage_Offset(Read_ISMC_Index(Head))) :=
            Write_ISMC_Token_Type(T_SM_Control);
        ISMC(Storage_Offset(Read_ISMC_Index(Head))+1) :=
            Write_ISMC_Control_Item(Item);
        Head := Write_ISMC_Index(Read_ISMC_Index(Head)+2);
    end;
```

```

   procedure Step is
   -- Precondition: Interpreter_State is assigned a valid state.
   -- Processes the actions defined for the given state, updates
   -- Interpreter_State to the state resulting from the transition conditions.
   -- Postcondition: Interpreter_State is assigned a valid state, machine heads
   -- are in consistent positions, or failure state and inconsistent heads.
   begin

      Interpreter_State := Get_Interpreter_State;

      case Interpreter_State is

         when Initial =>

            Interpreter_State := Initialize;

         when Initialize =>

            ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1) :=
               ISMC(INITIAL_STATE_REF_OFFSET..INITIAL_STATE_REF_OFFSET+1);
            ISMC(CURRENT_LEVEL_OFFSET) := 1;
            Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));

            if Token_Type = T_SM_Control and Token_SM_Control = SIMPLE_STATE then
               Interpreter_State := Enter_Simple_State;
            else
               Interpreter_State := Failure;
            end if;

         when Enter_Simple_State =>

            ISMC(EXIT_ACTION_MARKERS_OFFSET
               +(Storage_Count(ISMC(CURRENT_LEVEL_OFFSET)))*2
                 ..EXIT_ACTION_MARKERS_OFFSET
                   +(Storage_Count(ISMC(CURRENT_LEVEL_OFFSET)))*2+1) :=
               Write_ISMC_Index(UNDEFINED);
            ISMC(TC_HEAD_OFFSET..TC_HEAD_OFFSET+1) := Write_ISMC_Index(UNDEFINED);
            Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));

            If Token_Type = T_State_ID then
               Interpreter_State := Set_Active_Simple_State;
            else
               Interpreter_State := Failure;
            end if;

         when Set_Active_Simple_State =>

            ISMC(ACTIVE_STATE_OFFSET+Storage_Count(ISMC(CURRENT_LEVEL_OFFSET))) :=
               Token_State_ID;
            Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));

            if Token_Type = T_SM_Control then
               if Token_SM_Control = STOP then
                  Interpreter_State := Final;
               else
                  if Token_SM_Control = ENTRY_ACTIONS then
                     Interpreter_State := Process_SS_Entry_Actions;
                  else
                     if Token_SM_Control = EXIT_ACTIONS then
                        Interpreter_State := Set_SS_Exit_Action_Marker;
                     else
               if Token_SM_Control = TRANSITIONS then
                  Interpreter_State := Iterate_Transitions;
               else
                  Interpreter_State := Failure;
               end if;
            end if;
         end if;
      end if;
   else
      Interpreter_State := Failure;
   end if;

when Process_SS_Entry_Actions =>

   Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));

   if Token_Type = T_SM_Control then
      if Token_SM_Control = NATIVE_ACTION then
         Interpreter_State := Process_SS_Native_Entry_Action;
      else
         if Token_SM_Control = EXTERNAL_ACTION then
            Interpreter_State := Process_SS_External_Entry_Action;
         else
            if Token_SM_Control = EXIT_ACTIONS then
               Interpreter_State := Set_SS_Exit_Action_Marker;
            else
               if Token_SM_Control = TRANSITIONS then
                  Interpreter_State := Iterate_Transitions;
               else
                  Interpreter_State := Failure;
               end if;
            end if;
         end if;
      end if;
   else
      Interpreter_State := Failure;
   end if;

when Process_SS_Native_Entry_Action =>

   Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));

   if Token_Type = T_SM_Control then
      if Token_SM_Control = ASSIGN then
         Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));
         ISMC(ASSIGNMENT_MARKER_OFFSET..ASSIGNMENT_MARKER_OFFSET+1) :=
            Write_ISMC_Index(Token_ISMC_Index);
         Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));
         ISMC(Storage_Offset(Read_ISMC_Index(ISMC(
            ASSIGNMENT_MARKER_OFFSET..ASSIGNMENT_MARKER_OFFSET+1))))
            := Token_Byte;
      else
         if Token_SM_Control = SEND then
            Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));
            ISMC(TMP_MESSAGE_BUFFER_OFFSET..TMP_MESSAGE_BUFFER_OFFSET
               +HIRTE_Configuration.MSG_BUFFER_SIZE_BYTES-1) :=
               Token_Event;
         else
            Interpreter_State := FAILURE;
         end if;
      end if;
   else
```

```
                Interpreter_State := FAILURE;
            end if;

        when Process_SS_External_Entry_Action =>

            ISMC(EXTERNAL_OPERATION_MARKER_OFFSET
                ..EXTERNAL_OPERATION_MARKER_OFFSET+1) :=
                ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1);
            ISMC(EXTERNAL_OPERATION_MARKER_OFFSET
                ..EXTERNAL_OPERATION_MARKER_OFFSET+1) :=
                Write_ISMC_Index(Read_ISMC_Index(
                    ISMC(EXTERNAL_OPERATION_MARKER_OFFSET
                        ..EXTERNAL_OPERATION_MARKER_OFFSET+1))+1);
            Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));
            Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));

            if Token_Type = T_ISMC_Index then
                Interpreter_State := Set_Head_on_Next_SSEA_Item;
            else
                Interpreter_State := Failure;
            end if;

        when Set_Head_on_Next_SSEA_Item =>

            ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1) :=
                Write_ISMC_Index(Read_ISMC_Index(ISMC(I_HEAD_OFFSET
                    ..I_HEAD_OFFSET+1))+Token_ISMC_Index);

            if Read_ISMC_Index(ISMC(EXTERNAL_OPERATION_MARKER_OFFSET
                ..EXTERNAL_OPERATION_MARKER_OFFSET+1)) = COMPLETE then
                Interpreter_State := Process_SS_Entry_Actions;
            else
                Interpreter_State := Failure;
            end if;

        when Set_SS_Exit_Action_Marker =>

            ISMC(EXIT_ACTION_MARKERS_OFFSET
                +(Storage_Count(ISMC(CURRENT_LEVEL_OFFSET)))*2
                    ..EXIT_ACTION_MARKERS_OFFSET
                        +(Storage_Count(ISMC(CURRENT_LEVEL_OFFSET)))*2+1) :=
                ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1);
            ISMC(EXIT_ACTION_MARKERS_OFFSET
                +(Storage_Count(ISMC(CURRENT_LEVEL_OFFSET)))*2
                    ..EXIT_ACTION_MARKERS_OFFSET
                        +(Storage_Count(ISMC(CURRENT_LEVEL_OFFSET)))*2+1) :=
                Write_ISMC_Index(Read_ISMC_Index(ISMC(EXIT_ACTION_MARKERS_OFFSET
                    +(Storage_Count(ISMC(CURRENT_LEVEL_OFFSET)))*2
                        ..EXIT_ACTION_MARKERS_OFFSET
                            +(Storage_Count(ISMC(CURRENT_LEVEL_OFFSET)))*2+1))+3);
            Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));

            if Token_Type = T_ISMC_Index then
                Interpreter_State := Set_Head_on_Next_SS_Item;
            else
                Interpreter_State := Failure;
            end if;

        when Set_Head_on_Next_SS_Item =>

            ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1) :=
                Write_ISMC_Index(Read_ISMC_Index(ISMC(I_HEAD_OFFSET
                 ..I_HEAD_OFFSET+1))+Token_ISMC_Index);
            Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));

            if Token_Type = T_SM_Control and Token_SM_Control = TRANSITIONS then
               Interpreter_State := Iterate_Transitions;
            else
               Interpreter_State := Failure;
            end if;

         when Iterate_Transitions =>

            ISMC(INDEX_MARKER_OFFSET..INDEX_MARKER_OFFSET+1) :=
               ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1);
            Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));

            if Token_Type = T_SM_Control and Token_SM_Control = TRANSITION then
               Interpreter_State := Check_Transition;
            else
               Interpreter_State := Failure;
            end if;

         when Check_Transition =>

            ISMC(TMP_MESSAGE_BUFFER_OFFSET..TMP_MESSAGE_BUFFER_OFFSET
               +HIRTE_Configuration.MSG_BUFFER_SIZE_BYTES-1) := UNDEFINED_MSG;
            Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));

            if Token_Type = T_SM_Control then
               if Token_SM_Control = EVENT then
                  Interpreter_State := Check_Event;
               else
                  if Token_SM_Control = GUARD then
                     Interpreter_State := Check_Guard;
                  else
                     if Token_SM_Control = TC_MARKER then
                        Interpreter_State := Read_Transition_Chain_Marker;
                     else
                        Interpreter_State := Failure;
                     end if;
                  end if;
               end if;
            else
               Interpreter_State := Failure;
            end if;

         when Check_Event =>

            Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));

            if Token_Type = T_Event then
               Interpreter_State := Compare_Events;
            else
               Interpreter_State := Failure;
            end if;

         when Compare_Events =>

            if Token_Event = ISMC(LAST_MESSAGE_BUFFER_OFFSET
               ..LAST_MESSAGE_BUFFER_OFFSET
                  +HIRTE_Configuration.MSG_BUFFER_SIZE_BYTES-1) then
               Interpreter_State := Event_Occured;
            else
```

```
                Interpreter_State := Skip_Guard;
            end if;

        when Event_Occured =>

            ISMC(TMP_MESSAGE_BUFFER_OFFSET..TMP_MESSAGE_BUFFER_OFFSET
                +HIRTE_Configuration.MSG_BUFFER_SIZE_BYTES-1) := Token_Event;
            Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));

            if Token_Type = T_SM_Control then
                if Token_SM_Control = GUARD then
                    Interpreter_State := Check_Guard;
                else
                    if Token_SM_Control = TC_MARKER then
                        Interpreter_State := Read_Transition_Chain_Marker;
                    else
                        Interpreter_State := Failure;
                    end if;
                end if;
            else
                Interpreter_State := Failure;
            end if;

        when Check_Guard =>

            ISMC(STATUS_OFFSET) := Write_ISMC_Status(STATUS_UNDEFINED);
            Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));

            if Token_Type = T_SM_Control then
                if Token_SM_Control = NATIVE_GUARD then
                    Interpreter_State := Evaluate_Native_Guard;
                else
                    if Token_SM_Control = EXTERNAL_GUARD then
                        Interpreter_State := Evaluate_External_Guard;
                    else
                        Interpreter_State := Failure;
                    end if;
                end if;
            else
                Interpreter_State := Failure;
            end if;

        when Evaluate_Native_Guard =>

            Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));
            if Token_Type = T_ISMC_Index then
                ISMC(INTEGER_REGISTER_OFFSET..INTEGER_REGISTER_OFFSET+3) :=
                    ISMC(Storage_Offset(Token_ISMC_Index)
                        ..Storage_Offset(Token_ISMC_Index)+3);
            else
                Interpreter_State := Failure;
            end if;
            Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));
            if Token_Type = T_Integer then
                if Token_Integer = Read_Integer_Buffer(
                    ISMC(INTEGER_REGISTER_OFFSET..INTEGER_REGISTER_OFFSET+3)) then
                    ISMC(STATUS_OFFSET) := Write_ISMC_Status(GUARD_TRUE);
                    Interpreter_State := Firing_Condition_True;
                else
                    ISMC(STATUS_OFFSET) := Write_ISMC_Status(GUARD_FALSE);
                    Interpreter_State := Firing_Condition_False;
                end if;
```

```
          else
             Interpreter_State := Failure;
          end if;

       when Skip_Guard =>

          Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));

          if Token_Type = T_SM_Control then
             if Token_SM_Control = TC_MARKER then
                Interpreter_State := Skip_Transition_Chain_Marker;
             else
                if Token_SM_Control = GUARD then
                   Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));
                   if Token_SM_Control = NATIVE_GUARD then
                      Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));
                      Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));
                      Interpreter_State := Firing_Condition_False;
                   else
                      if Token_SM_Control = EXTERNAL_GUARD then
                         Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));
                         Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));
                         ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1) :=
                            Write_ISMC_Index(Read_ISMC_Index(ISMC(I_HEAD_OFFSET
                               ..I_HEAD_OFFSET+1))+Token_ISMC_Index);
                         Interpreter_State := Firing_Condition_False;
                      else
                         Interpreter_State := Failure;
                      end if;
                   end if;
                else
                   Interpreter_State := Failure;
                end if;
             end if;
          else
             Interpreter_State := Failure;
          end if;

       when Evaluate_External_Guard =>

          ISMC(EXTERNAL_OPERATION_MARKER_OFFSET
             ..EXTERNAL_OPERATION_MARKER_OFFSET+1) :=
             ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1);
          ISMC(EXTERNAL_OPERATION_MARKER_OFFSET
             ..EXTERNAL_OPERATION_MARKER_OFFSET+1) :=
             Write_ISMC_Index(Read_ISMC_Index(ISMC(
                EXTERNAL_OPERATION_MARKER_OFFSET
                   ..EXTERNAL_OPERATION_MARKER_OFFSET+1))+1);
          Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));
          Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));

          if Token_Type = T_ISMC_Index then
             Interpreter_State := Set_Head_on_Next_Guard_Item;
          else
             Interpreter_State := Failure;
          end if;

       when Set_Head_on_Next_Guard_Item =>

          ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1) :=
             Write_ISMC_Index(Read_ISMC_Index(ISMC(I_HEAD_OFFSET
                ..I_HEAD_OFFSET+1))+Token_ISMC_Index);

          if Read_ISMC_Index(ISMC(EXTERNAL_OPERATION_MARKER_OFFSET
             ..EXTERNAL_OPERATION_MARKER_OFFSET+1)) = COMPLETE then
             if Read_ISMC_Status(ISMC(STATUS_OFFSET)) = GUARD_TRUE then
                Interpreter_State := Firing_Condition_True;
             else
                Interpreter_State := Firing_Condition_False;
             end if;
          else
             Interpreter_State := Failure;
          end if;

       when Firing_Condition_False =>

          Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));

          if Token_Type = T_SM_Control and Token_SM_Control = TC_MARKER then
             Interpreter_State := Skip_Transition_Chain_Marker;
          else
             Interpreter_State := Failure;
          end if;

       when Firing_Condition_True =>

          Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));

          if Token_Type = T_SM_Control and Token_SM_Control = TC_MARKER then
             Interpreter_State := Read_Transition_Chain_Marker;
          else
             Interpreter_State := Failure;
          end if;

       when Skip_Transition_Chain_Marker =>

          Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));
          Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));

          if Token_Type = T_State_ID then
             Interpreter_State := Skip_Transition_Source;
          else
             Interpreter_State := Failure;
          end if;

       when Read_Transition_Chain_Marker =>

          Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));

          if Token_Type = T_ISMC_Index then
             Interpreter_State := Set_Transition_Chain_Head;
          else
             Interpreter_State := Failure;
          end if;

       when Set_Transition_Chain_Head =>

          ISMC(TC_HEAD_OFFSET..TC_HEAD_OFFSET +1) :=
             Write_ISMC_Index(Token_ISMC_Index);
          ISMC(TRANSITION_TRIGGER_OFFSET..TRANSITION_TRIGGER_OFFSET
             +HIRTE_Configuration.MSG_BUFFER_SIZE_BYTES-1) :=
             ISMC(TMP_MESSAGE_BUFFER_OFFSET..TMP_MESSAGE_BUFFER_OFFSET
                +HIRTE_Configuration.MSG_BUFFER_SIZE_BYTES-1);
          Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));
```

```

         if Token_Type = T_State_ID then
            Interpreter_State := Set_Transition_Source;
         else
            Interpreter_State := Failure;
         end if;

      when Skip_Transition_Source =>

         Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));

         if Token_Type = T_ISMC_Index then
            Interpreter_State := Determine_Triggering;
         else
            Interpreter_State := Failure;
         end if;

      when Check_Next_Transition =>

         ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1) :=
            Write_ISMC_Index(Token_ISMC_Index);
         Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));

         if Token_Type = T_SM_Control and Token_SM_Control = TRANSITION then
            Interpreter_State := Check_Transition;
         else
            Interpreter_State := Failure;
         end if;

      when Set_Transition_Source =>

         ISMC(TRANSITION_SOURCE_OFFSET) := Token_State_ID;
         Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));

         if Token_Type = T_ISMC_Index then
            Interpreter_State := Determine_Triggering;
         else
            Interpreter_State := Failure;
         end if;

      when Reiterate_Transitions =>

         ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1) :=
            ISMC(INDEX_MARKER_OFFSET..INDEX_MARKER_OFFSET+1);
         Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));

         if Token_Type = T_SM_Control and Token_SM_Control = TRANSITION then
            Interpreter_State := Check_Transition;
         else
            Interpreter_State := Failure;
         end if;

      when Determine_Triggering =>

         if Token_ISMC_Index = UNDEFINED then
            if Read_ISMC_Index(ISMC(TC_HEAD_OFFSET..TC_HEAD_OFFSET+1))
               = UNDEFINED then
               Interpreter_State := Reiterate_Transitions;
            else
               Interpreter_State := Leave_Simple_State;
            end if;
         else
```

```
                Interpreter_State := Check_Next_Transition;
            end if;

        when Set_Head_on_SS_Exit_Actions =>

            ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1) :=
                ISMC(EXIT_ACTION_MARKERS_OFFSET
                    +(Storage_Count(ISMC(CURRENT_LEVEL_OFFSET)))*2
                        ..EXIT_ACTION_MARKERS_OFFSET
                            +(Storage_Count(ISMC(CURRENT_LEVEL_OFFSET)))*2+1);

            Interpreter_State := Process_SS_Exit_Actions;

        when Process_SS_Exit_Actions =>

            Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));

            if Token_Type = T_SM_Control then
                if Token_SM_Control = NATIVE_ACTION then
                    Interpreter_State := Process_SS_Native_Exit_Action;
                else
                    if Token_SM_Control = EXTERNAL_ACTION then
                        Interpreter_State := Process_SS_External_Exit_Action;
                    else
                        if Token_SM_Control = EOL then
                            Interpreter_State := Determine_Triggering_Event;
                        else
                            Interpreter_State := Failure;
                        end if;
                    end if;
                end if;
            else
                Interpreter_State := Failure;
            end if;

        when Process_SS_Native_Exit_Action =>

            -- See above.

        when Process_SS_External_Exit_Action =>

            ISMC(EXTERNAL_OPERATION_MARKER_OFFSET
                ..EXTERNAL_OPERATION_MARKER_OFFSET+1) :=
                ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1);
            ISMC(EXTERNAL_OPERATION_MARKER_OFFSET
                ..EXTERNAL_OPERATION_MARKER_OFFSET+1) :=
                Write_ISMC_Index(Read_ISMC_Index(ISMC(
                    EXTERNAL_OPERATION_MARKER_OFFSET
                        ..EXTERNAL_OPERATION_MARKER_OFFSET+1))+1);
            Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));
            Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));

            if Token_Type = T_ISMC_Index then
                Interpreter_State := Set_Head_on_Next_SSXA_Item;
            else
                Interpreter_State := Failure;
            end if;

        when Set_Head_on_Next_SSXA_Item =>

            ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1) :=
                Write_ISMC_Index(Read_ISMC_Index(ISMC(I_HEAD_OFFSET
                    ..I_HEAD_OFFSET+1))+Token_ISMC_Index);

                if Read_ISMC_Index(ISMC(EXTERNAL_OPERATION_MARKER_OFFSET
                    ..EXTERNAL_OPERATION_MARKER_OFFSET+1)) = COMPLETE then
                    Interpreter_State := Process_SS_Exit_Actions;
                else
                    Interpreter_State := Failure;
                end if;

            when Leave_Simple_State =>

                if Read_ISMC_Index(ISMC(EXIT_ACTION_MARKERS_OFFSET
                    +(Storage_Count(ISMC(CURRENT_LEVEL_OFFSET)))*2
                        ..EXIT_ACTION_MARKERS_OFFSET
                            +(Storage_Count(ISMC(CURRENT_LEVEL_OFFSET)))*2+1))
                    /= UNDEFINED then
                    Interpreter_State := Set_Head_on_SS_Exit_Actions;
                else
                    Interpreter_State := Determine_Triggering_Event;
                end if;

            when Determine_Triggering_Event =>

                if ISMC(LAST_MESSAGE_BUFFER_OFFSET..LAST_MESSAGE_BUFFER_OFFSET
                    +HIRTE_Configuration.MSG_BUFFER_SIZE_BYTES-1)
                        = ISMC(TRANSITION_TRIGGER_OFFSET..TRANSITION_TRIGGER_OFFSET
                            +HIRTE_Configuration.MSG_BUFFER_SIZE_BYTES-1)
                                and ISMC(TRANSITION_TRIGGER_OFFSET
                                    ..TRANSITION_TRIGGER_OFFSET
                                        +HIRTE_Configuration.MSG_BUFFER_SIZE_BYTES-1)
                    /= UNDEFINED_MSG then
                    Interpreter_State := Consume_Message;
                else
                    Interpreter_State := Reach_Transition_Level;
                end if;

            when Consume_Message =>

                ISMC(LAST_MESSAGE_BUFFER_OFFSET..LAST_MESSAGE_BUFFER_OFFSET
                    +HIRTE_Configuration.MSG_BUFFER_SIZE_BYTES-1) := UNDEFINED_MSG;

                Interpreter_State := Reach_Transition_Level;

            when Reach_Transition_Level =>

                if ISMC(ACTIVE_STATE_OFFSET+Storage_Count(ISMC(CURRENT_LEVEL_OFFSET)))
                    = ISMC(TRANSITION_SOURCE_OFFSET) then
                    Interpreter_State := Process_Transition;
                else
                    Interpreter_State := Determine_HLT_Exit_Actions;
                end if;

                ISMC(ACTIVE_STATE_OFFSET+Storage_Count(ISMC(CURRENT_LEVEL_OFFSET))) :=
                    UNDEFINED_STATE;

            when Determine_HLT_Exit_Actions =>

                ISMC(CURRENT_LEVEL_OFFSET) := ISMC(CURRENT_LEVEL_OFFSET)-1;

                if Read_ISMC_Index(ISMC(EXIT_ACTION_MARKERS_OFFSET
                    +(Storage_Count(ISMC(CURRENT_LEVEL_OFFSET)))*2
                        ..EXIT_ACTION_MARKERS_OFFSET
                    +(Storage_Count(ISMC(CURRENT_LEVEL_OFFSET)))*2+1))
                /= UNDEFINED then
                Interpreter_State := Set_Head_on_HLT_Exit_Actions;
            else
                Interpreter_State := Leave_HLT_Composite_State;
            end if;

        when Process_Transition =>

            Read_Tape(ISMC(TC_HEAD_OFFSET..TC_HEAD_OFFSET+1));

            if Token_Type = T_SM_Control then
                if Token_SM_Control = TARGETS then
                    Interpreter_State := Transit;
                else
                    if Token_SM_Control = TRANSITION_ACTIONS then
                        Interpreter_State := Process_Transition_Actions;
                    else
                        Interpreter_State := Failure;
                    end if;
                end if;
            else
                Interpreter_State := Failure;
            end if;

        when Set_Head_on_HLT_Exit_Actions =>

            ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1) :=
                ISMC(EXIT_ACTION_MARKERS_OFFSET
                    +(Storage_Count(ISMC(CURRENT_LEVEL_OFFSET)))*2
                        ..EXIT_ACTION_MARKERS_OFFSET
                            +(Storage_Count(ISMC(CURRENT_LEVEL_OFFSET)))*2+1);

            Interpreter_State := Process_HLT_Exit_Actions;

        when Process_HLT_Exit_Actions =>

            Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));

            if Token_Type = T_SM_Control then
                if Token_SM_Control = NATIVE_ACTION then
                    Interpreter_State := Process_HLT_Native_Exit_Action;
                else
                    if Token_SM_Control = EXTERNAL_ACTION then
                        Interpreter_State := Process_HLT_External_Exit_Action;
                    else
                        if Token_SM_Control = EOL then
                            Interpreter_State := Leave_HLT_Composite_State;
                        else
                            Interpreter_State := Failure;
                        end if;
                    end if;
                end if;
            else
                Interpreter_State := Failure;
            end if;

        when Process_HLT_Native_Exit_Action =>

            -- See above.

        when Process_HLT_External_Exit_Action =>
```

```

          ISMC(EXTERNAL_OPERATION_MARKER_OFFSET
              ..EXTERNAL_OPERATION_MARKER_OFFSET+1) :=
             ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1);
          ISMC(EXTERNAL_OPERATION_MARKER_OFFSET
              ..EXTERNAL_OPERATION_MARKER_OFFSET+1) :=
             Write_ISMC_Index(Read_ISMC_Index(ISMC(
                EXTERNAL_OPERATION_MARKER_OFFSET
                    ..EXTERNAL_OPERATION_MARKER_OFFSET+1))+1);
          Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));
          Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));

          if Token_Type = T_ISMC_Index then
             Interpreter_State := Set_Head_on_Next_HLTXA_Item;
          else
             Interpreter_State := Failure;
          end if;

       when Set_Head_on_Next_HLTXA_Item =>

          ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1) :=
             Write_ISMC_Index(Read_ISMC_Index(ISMC(I_HEAD_OFFSET
                 ..I_HEAD_OFFSET+1))+Token_ISMC_Index);

          if Read_ISMC_Index(ISMC(EXTERNAL_OPERATION_MARKER_OFFSET
             ..EXTERNAL_OPERATION_MARKER_OFFSET+1)) = COMPLETE then
             Interpreter_State := Process_HLT_Exit_Actions;
          else
             Interpreter_State := Failure;
          end if;

       when Leave_HLT_Composite_State =>

          if ISMC(CURRENT_LEVEL_OFFSET) < 1 then
             Interpreter_State := Set_Head_on_Chain;
          else
             if ISMC(ACTIVE_STATE_OFFSET
                +Storage_Count(ISMC(CURRENT_LEVEL_OFFSET)))
                   = ISMC(TRANSITION_SOURCE_OFFSET) then
                Interpreter_State := Process_Transition;
             else
                Interpreter_State := Determine_HLT_Exit_Actions;
             end if;
          end if;

          ISMC(ACTIVE_STATE_OFFSET
             +Storage_Count(ISMC(CURRENT_LEVEL_OFFSET))) := UNDEFINED_STATE;

       when Transit =>

          Read_Tape(ISMC(TC_HEAD_OFFSET..TC_HEAD_OFFSET+1));

          if Token_Type = T_ISMC_Index then
             Interpreter_State := Set_Head_on_State;
          else
             Interpreter_State := Failure;
          end if;

       when Process_Transition_Actions =>

          Read_Tape(ISMC(TC_HEAD_OFFSET..TC_HEAD_OFFSET+1));

```

```
            if Token_Type = T_SM_Control then
               if Token_SM_Control = NATIVE_ACTION then
                  Interpreter_State := Process_Native_Transition_Action;
               else
                  if Token_SM_Control = EXTERNAL_ACTION then
                     Interpreter_State := Process_External_Transition_Action;
                  else
                     if Token_SM_Control = TARGETS then
                        Interpreter_State := Transit;
                     else
                        Interpreter_State := Failure;
                     end if;
                  end if;
               end if;
            else
               Interpreter_State := Failure;
            end if;

         when Process_Native_Transition_Action =>

            -- See above.

         when Process_External_Transition_Action =>

            ISMC(EXTERNAL_OPERATION_MARKER_OFFSET
               ..EXTERNAL_OPERATION_MARKER_OFFSET+1) :=
               ISMC(TC_HEAD_OFFSET..TC_HEAD_OFFSET+1);
            ISMC(EXTERNAL_OPERATION_MARKER_OFFSET
               ..EXTERNAL_OPERATION_MARKER_OFFSET+1) :=
               Write_ISMC_Index(Read_ISMC_Index(ISMC(
                  EXTERNAL_OPERATION_MARKER_OFFSET
                     ..EXTERNAL_OPERATION_MARKER_OFFSET+1))+1);
            Read_Tape(ISMC(TC_HEAD_OFFSET..TC_HEAD_OFFSET+1));
            Read_Tape(ISMC(TC_HEAD_OFFSET..TC_HEAD_OFFSET+1));

            if Token_Type = T_ISMC_Index then
               Interpreter_State := Set_Head_on_Next_TA_Item;
            else
               Interpreter_State := Failure;
            end if;

         when Set_Head_on_Next_TA_Item =>

            ISMC(TC_HEAD_OFFSET..TC_HEAD_OFFSET+1) :=
               Write_ISMC_Index(Read_ISMC_Index(ISMC(
                  TC_HEAD_OFFSET..TC_HEAD_OFFSET+1))+Token_ISMC_Index);

            if Read_ISMC_Index(ISMC(EXTERNAL_OPERATION_MARKER_OFFSET
               ..EXTERNAL_OPERATION_MARKER_OFFSET+1)) = COMPLETE then
               Interpreter_State := Process_Transition_Actions;
            else
               Interpreter_State := Failure;
            end if;

         when Set_Head_on_State =>

            ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1) :=
               Write_ISMC_Index(Token_ISMC_Index);
            Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));

            if Token_Type = T_SM_Control then
               if Token_SM_Control = SIMPLE_STATE then
```

```
                Interpreter_State := Enter_Simple_State;
            else
                if Token_SM_Control = COMPOSITE_STATE then
                    Interpreter_State := Process_Composite_State;
                else
                    if Token_SM_Control = DSM_FRAGMENT then
                        Interpreter_State := Process_DSM_Fragment;
                    else
                        Interpreter_State := Failure;
                    end if;
                end if;
            end if;
        else
            Interpreter_State := Failure;
        end if;

    when Process_Composite_State =>

        Read_Tape(ISMC(TC_HEAD_OFFSET..TC_HEAD_OFFSET+1));

        if Token_Type = T_SM_Control then
            if Token_SM_Control = INBOUND then
                Interpreter_State := Enter_Composite_State;
            else
                if Token_SM_Control = OUTBOUND then
                    Interpreter_State := Determine_CS_Exit_Actions;
                else
                    Interpreter_State := Failure;
                end if;
            end if;
        else
            Interpreter_State := Failure;
        end if;

    when Enter_Composite_State =>

        ISMC(EXIT_ACTION_MARKERS_OFFSET
            +(Storage_Count(ISMC(CURRENT_LEVEL_OFFSET)))*2
                ..EXIT_ACTION_MARKERS_OFFSET
                    +(Storage_Count(ISMC(CURRENT_LEVEL_OFFSET)))*2+1) :=
            Write_ISMC_Index(UNDEFINED);

        Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));

        if Token_Type = T_State_ID then
            Interpreter_State := Set_Active_Composite_State;
        else
            Interpreter_State := Failure;
        end if;

    when Set_Active_Composite_State =>

        ISMC(ACTIVE_STATE_OFFSET
            +Storage_Count(ISMC(CURRENT_LEVEL_OFFSET))) := Token_State_ID;
        Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));

        if Token_Type = T_SM_Control then
            if Token_SM_Control = ENTRY_ACTIONS then
                Interpreter_State := Process_CS_Entry_Actions;
            else
                if Token_SM_Control = EXIT_ACTIONS then
                    Interpreter_State := Set_CS_Exit_Action_Marker;
                    else
                        if Token_SM_Control = EOL then
                            Interpreter_State := Raise_Hierarchy_Level;
                        else
                            Interpreter_State := Failure;
                        end if;
                    end if;
                end if;
            else
                Interpreter_State := Failure;
            end if;

        when Process_CS_Entry_Actions =>

            Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));

            if Token_Type = T_SM_Control then
                if Token_SM_Control = NATIVE_ACTION then
                    Interpreter_State := Process_CS_Native_Entry_Action;
                else
                    if Token_SM_Control = EXTERNAL_ACTION then
                        Interpreter_State := Process_CS_External_Entry_Action;
                    else
                        if Token_SM_Control = EXIT_ACTIONS then
                            Interpreter_State := Set_CS_Exit_Action_Marker;
                        else
                            if Token_SM_Control = EOL then
                                Interpreter_State := Raise_Hierarchy_Level;
                            else
                                Interpreter_State := Failure;
                            end if;
                        end if;
                    end if;
                end if;
            else
                Interpreter_State := Failure;
            end if;

        when Process_CS_Native_Entry_Action =>

            -- See above.

        when Process_CS_External_Entry_Action =>

            ISMC(EXTERNAL_OPERATION_MARKER_OFFSET
                ..EXTERNAL_OPERATION_MARKER_OFFSET+1) :=
                ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1);
            ISMC(EXTERNAL_OPERATION_MARKER_OFFSET
                ..EXTERNAL_OPERATION_MARKER_OFFSET+1) :=
                Write_ISMC_Index(Read_ISMC_Index(ISMC(
                    EXTERNAL_OPERATION_MARKER_OFFSET
                        ..EXTERNAL_OPERATION_MARKER_OFFSET+1))+1);
            Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));
            Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));

            if Token_Type = T_ISMC_Index then
                Interpreter_State := Set_Head_on_Next_CSEA_Item;
            else
                Interpreter_State := Failure;
            end if;

        when Set_Head_on_Next_CSEA_Item =>
```

```

            ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1) :=
                Write_ISMC_Index(Read_ISMC_Index(ISMC(I_HEAD_OFFSET
                    ..I_HEAD_OFFSET+1))+Token_ISMC_Index);

            if Read_ISMC_Index(ISMC(EXTERNAL_OPERATION_MARKER_OFFSET
                ..EXTERNAL_OPERATION_MARKER_OFFSET+1)) = COMPLETE then
                Interpreter_State := Process_CS_Entry_Actions;
            else
                Interpreter_State := Failure;
            end if;

        when Set_CS_Exit_Action_Marker =>

            ISMC(EXIT_ACTION_MARKERS_OFFSET
                +(Storage_Count(ISMC(CURRENT_LEVEL_OFFSET)))*2
                    ..EXIT_ACTION_MARKERS_OFFSET
                        +(Storage_Count(ISMC(CURRENT_LEVEL_OFFSET)))*2+1) :=
                ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1);

            Interpreter_State := Raise_Hierarchy_Level;

        when Raise_Hierarchy_Level =>

            ISMC(CURRENT_LEVEL_OFFSET) := ISMC(CURRENT_LEVEL_OFFSET)+1;

            Interpreter_State := Transit;

        when Determine_CS_Exit_Actions =>

            ISMC(CURRENT_LEVEL_OFFSET) := ISMC(CURRENT_LEVEL_OFFSET)-1;

            if Read_ISMC_Index(ISMC(EXIT_ACTION_MARKERS_OFFSET
                +(Storage_Count(ISMC(CURRENT_LEVEL_OFFSET)))*2
                    ..EXIT_ACTION_MARKERS_OFFSET
                        +(Storage_Count(ISMC(CURRENT_LEVEL_OFFSET)))*2+1))
                /= UNDEFINED then
                Interpreter_State := Set_Head_on_CS_Exit_Actions;
            else
                Interpreter_State := Leave_Composite_State;
            end if;

        when Set_Head_on_CS_Exit_Actions =>

            ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1) :=
              ISMC(EXIT_ACTION_MARKERS_OFFSET
                  +(Storage_Count(ISMC(CURRENT_LEVEL_OFFSET)))*2
                      ..EXIT_ACTION_MARKERS_OFFSET
                          +(Storage_Count(ISMC(CURRENT_LEVEL_OFFSET)))*2+1);

            Interpreter_State := Process_CS_Exit_Actions;

        when Process_CS_Exit_Actions =>

            Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));

            if Token_Type = T_SM_Control then
                if Token_SM_Control = NATIVE_ACTION then
                    Interpreter_State := Process_CS_Native_Exit_Action;
                else
                    if Token_SM_Control = EXTERNAL_ACTION then
                        Interpreter_State := Process_CS_External_Exit_Action;
                else
                    if Token_SM_Control = EOL then
                        Interpreter_State := Leave_Composite_State;
                    else
                        Interpreter_State := Failure;
                    end if;
                end if;
            end if;
        else
            Interpreter_State := Failure;
        end if;

    when Process_CS_Native_Exit_Action =>

        -- See above.

    when Process_CS_External_Exit_Action =>

        ISMC(EXTERNAL_OPERATION_MARKER_OFFSET
            ..EXTERNAL_OPERATION_MARKER_OFFSET+1) :=
            ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1);
        ISMC(EXTERNAL_OPERATION_MARKER_OFFSET
            ..EXTERNAL_OPERATION_MARKER_OFFSET+1) :=
            Write_ISMC_Index(Read_ISMC_Index(ISMC(
                EXTERNAL_OPERATION_MARKER_OFFSET
                    ..EXTERNAL_OPERATION_MARKER_OFFSET+1))+1);
        Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));
        Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));

        if Token_Type = T_ISMC_Index then
            Interpreter_State := Set_Head_on_Next_CSXA_Item;
        else
            Interpreter_State := Failure;
        end if;

    when Set_Head_on_Next_CSXA_Item =>

        ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1) :=
            Write_ISMC_Index(Read_ISMC_Index(ISMC(I_HEAD_OFFSET
                ..I_HEAD_OFFSET+1))+Token_ISMC_Index);

        if Read_ISMC_Index(ISMC(EXTERNAL_OPERATION_MARKER_OFFSET
            ..EXTERNAL_OPERATION_MARKER_OFFSET+1)) = COMPLETE then
            Interpreter_State := Process_CS_Exit_Actions;
        else
            Interpreter_State := Failure;
        end if;

    when Leave_Composite_State =>

        ISMC(ACTIVE_STATE_OFFSET
            +Storage_Count(ISMC(CURRENT_LEVEL_OFFSET))) := UNDEFINED_STATE;

        if ISMC(CURRENT_LEVEL_OFFSET) >= 1 then
            Interpreter_State := Read_Next_Target;
        else
            Interpreter_State := Outside_Machine_Scope;
        end if;

    when Final =>

        null;
```

```

               when Failure =>

                  null;

            end case;

            Set_Interpreter_State(Interpreter_State);

         end;

      end ISMC_Automaton;

end HIRTE_ISMC_Interpreter;
```

why case-when instead of SRSM

Listing 4.12 implements the ISMC interpreter. After discussing and dismissing established state machine implementation approaches and introducing an allegedly more efficient solution, it may appear contradictory to use a plain *case-when* construct. However, there are a number of valid reasons. First, the ISMC interpreter includes an example for a parser – it traverses a sequential data structure with the recognized tokens directly actuating the interpreter actions. A flat, cyclic command structure like *case-when* is a fitting answer. Our SRSM pattern rather aims at more complex, hierarchical activities. Second, beyond effectiveness at least in the sense of a proof of concept, the reference implementation is supposed to convey a clearly represented, unobfuscated algorithm, illustrating the pattern workings as actual Ada code. *Case-when* requires no state machine management overhead if applied to a flat automaton like the given, putting the states, transitions and their relations to the fore, thusly making it better suitable for exemplification.

concerning native and external ops

Another remark is indicated by the differentiation of *native* and *external* operations. A reason for this distinction was mentioned: native operations are implemented in the context of the generic machine interpreter. Their consequently generic functionality may – and potentially needs to – be complemented by specialized, e.g. domain-specific or optimized, operations of the application. Furthermore, native operations comply to the step-wise execution of the ISMC, i.e. they do not require special consideration concerning scheduling. In contrast, external operation invocation implies atomic *run-to-completion.* If the operation is not fragmented into fixed run-time subprocedures called by successive ISMC instructions, it will execute without regard to a scheduler[16].

For an example of a VCU *Run* procedure with external operation calls refer to listing 4.16.

4.2.3.6 Variants and Extensions

reducing ISMC memory demand

To keep the binary sequence lean, i.e. to avoid unnecessary control overhead, we did not select ASN.1 for specification and implementation of the ISMC tape. However, the *Packed Encoding Rules* (PER) of this standard suggest a

[16]A preemptive multitasking of some underlying OS notwithstanding.

sensible optimization of our defined format. For the conceptual scope of this work, bytewise encoding of all ISMC token types assists the comprehension of the structures, not to say tests and empirical validation – and also a human reader may gain insights from a glance at the byte stream and recognize the opcodes. For an actual deployment of the architecture, size of the ISMC programs may be reduced significantly by applying encoding rules similar to PER: e.g. for the state machine control instructions, 5 bits are sufficient (cmp. listing 4.11). As our provided Ada reference sources demonstrate, adaption of the internal memory representation is straightforward (the *'Size* and *'Alignment* attributes). With the corresponding basic modifications of the interpreter's *Read Tape* procedure, the system would benefit from the reduction.

activity monitoring

The addition of a *state tracer* (cmp. 4.2.2) to a VCU set enables a detailed and systematic monitoring of both the virtual machine and application activities. If all components report their states quasi real-time, a control instance may realize functionality beyond logging for after-action analysis. The introduction of configurable timing and other constraints related to the designated execution profile of a software based on HIRTE could provide immediate feedback to the user (MMI) or a center component (OTA). Especially an early warning distribution of these measurement results to the center would benefit stable operations, as preemptive action could be taken in the case of increasing deviation from regular parameters, e.g. concerning quality of service (cmp. the *suspicious incident* of 3.1.2). But also locally – considering the potentially unsafe environment of our components – the software could at least selectively request resources for proper execution.

4.2.3.7 Related Approaches

Mentionable works that at least bear a resemblance in name/concept or field of application are *DCharts* ([Fe04]) and *virtual finite state machines* (Vfsm, [WSWW06]).

DCharts

DCharts ambitiously cover a wide range of statechart elements, including history and concurrent composite states. Similar to our approach, they provide a description language with an interpreter and also a compiler, producing a state machine implementation in Java. In contrast however, the interpreter takes the role of a simulator, not an RTE for field operations, i.e. it consciously does not pursue efficiency of execution. This directly affects the model description format, which rather has an intermediary character; a view we dismissed in 4.2.3.2. While emphasizing performance, the generated Java machine representation is complex and relies on dynamic features. Applied to HIRTE, this would violate our HI restrictions.

Vfsm

Vfsm and their development suite StateWORKS establish a method based on a virtual environment. Consisting of name sets for inputs, outputs and states, it is different to the HIRTE, focusing on the specification of decisions as essence of a state machine. The definition of corresponding conditions for entry actions, transitions and exit actions takes the form of "positive logic algebra" expressions (because *NOT* is not applicable) on the name sets. For the overall state machine specification, [WSWW06] gives a state transition table concept on flat state sets that associates actions and transitions with the triggering conditions. All thusly created objects are organized in a real-time database (RTDB), which serves as a repository for the RTE, interpreting and executing the loaded structures. Vfsms do not support composite states, instead advocating master-slave-hierarchies of state machines. With their *OFUN* user defined functions, they provide a mechanism similar, while more limited, to the HIRTE external operations.

4.2.4 Distributable State Machine Fragment Pattern

4.2.4.1 Abstract

revisiting SC interoperability

In chapter 3.1.3.3, we suggested a smart card integrated interoperability solution. An essential feature of that scenario was the partitioning of computation. The smart card, a secure but very limited platform, delegated complex procedures and sensor data evaluation to an external RTE. To a better part, this was security-driven: the operator may keep proprietary algorithms and business process implementations on the card; like private keys, they are not meant for dissemination.

relation to VCU scalability

Now, ISMC and VCUs present a similar constellation that allows us a generalization of the use case, i.e. broadening the applicability scope, and to provide an actual realization of the concept. In the previous sections, structure and reference implementation of the ISMC interpreter illustrated the scalability of the VCU. Depending on the intended run-time platform, the interpreter automaton may immediately provide computation-heavy native operations, or it may resort to a comprehensive external operation set implemented by its associated component. Alternatively, the VCU may be reduced to a minimum, resulting – as shown – in a lean engine with static memory requirements for traversing state machines in a safe and controlled manner.

moving beyond RPCs

Apparently, these two extremes – a fat versus thin RTE – reflect our initial scenario. Still, the external processing mechanism of section 4.2.3 applies locally to the VCU RTE and is restricted to procedure calls. Consequently, this section complements the VCU with a means for distributed machine execution. Due to the nature of the domain of this work, remote processing needs to go beyond remote procedure calls (RPC): aside from security issues like above, we have to presume that the remote system provides unique, i.e. locally not available, but generic functionality that is meant to be employed by specific business logics, and furthermore that these logics might include a number of calls too high for efficient exchange between RTEs. Instead, the remote RTE receives instructions on how to compute a replyable result with its proprietary operations or other capabilities. With this, HIRTE moves from simply invoking a procedure implemented on a remote RTE to the external execution with a preparatory selective outsourcing and subsequent return of the program fragment itself.

ISMC adaption

Consolidating the statements, HIRTE requires a mechanism to coherently delegate complex computations to remote RTEs, while upholding security, e.g. privacy of its application activities. Section 4.1.3 delivers the appropriate concept: applied to ISMC as *distributable state machine fragment* (DSMF), the *self-contained state* enables selective delegation of machine parts.

4.2.4.2 Extending the ISMC

The introduction of DSMFs to ISMC implies extending the ISMC syntax and enabling the interpreter to handle DSMFs accordingly. Listing 4.13 introduces the DSMF to the ISMC syntax. Productions directly refer to listing 4.10, integrating the extensions into the existing structures.

Listing 4.13: ISMC DSMF Extensions in EBNF

```
Header = ... , ISMC Program Length , ...
ISMC Program Length = ISMC Index ;
...

Registers = ...
             DSM Fragment Marker ,
             DSMF Transition Chain ,
             ...
DSM Fragment Marker = ISMC Index ;
DSMF Transition Chain = m * Target ; (* Depending on hierarchy depth *)
...

State = Simple State | Composite State | DSM Fragment;
DSM Fragment = DSM FRAGMENT , Requirements , Interpretable State Machine Code;
Requirements = Variable Subset , Operation Requirements , Event Requirements ;
Variable Subset = { Variable Reference Pair } , EOL ;
Variable Reference Pair = SM Variable , Fragment Variable ;
Operation Requirements = ? Application-specific ? ;
Event Requirements = ? Application-specific ? ;
SM Variable = ISMC Index ;
Fragment Variable = ISMC Index ;
```

header extension

The ISMC *Header* adds the *ISMC Program Length*, indicating the overall byte usage on an ISMC tape. Additional *Registers* are a *DSM Fragment Marker* – specifies the starting index of the head's current fragment, similar to the *External Operation Marker* – and the *DSMF Transition Chain*, to store an inbound sequence of states in a fragment composite state, used by the remote interpreter on entering a fragment.

state extension

As a complementing *State*, the *DSM Fragment* consists of *Requirements* and, consistently, the structure of a complete *Interpretable State Machine Code*. This recursive inclusion achieves orthogonality of the ISMC and fragment concepts in an inornate way.

The *Variable Subset* declares a mapping between state machine variables and fragment variables. These are referenced by their respective indices in the ISMC byte array. In the reference implementation, for demonstration and brevity purposes, the interpreter assumes integer variables of four byte length. An actual application will consider additional type differentiation, e.g. by TLV-encoding. Alternatively, to keep variable memory consumption minimal and in line with the given record representation (see sources above), it might implement an indiscriminate byte-wise mapping. Moreover, specification of *Operation* and *Event Requirements*, and the corresponding checks, fully depend on conventions of the remote RTE and its application. Thus, while being significant to the ISMC definition, they are beyond the interpreter's scope, which merely needs to provide a referenceable, structured container for the requirements, as given.

interpreter extension

Figure 4.15 specifies the necessary extensions of the ISMC interpreter automaton to handle – especially hand over, maintaining coherency – DSM fragments. The greyed states represent established parts of the automaton as given by figures 4.9 to 4.12, with their existing transitions omitted, complemented by new relations.

fragment conventions

Note that while the above syntax structure of the fragment complies with the definition of an ISMC, the interpreter automaton still enables the state machine to treat a fragment like any composite state, including direct transitions into and out of the fragment state hierarchy. Implied is a handover mechanism that touches a number of semantic variation points. For example, the reference implementation presumes that events that trigger transitions with the fragment state itself as source have to occur in the remote RTE, equal to transitions inside the fragment. Furthermore, message respectively event buffers are currently not included in the handover – other realizations might take a different view.

4.2.4.3 Structure

Figure 4.16 presents the structure of the DSMF pattern.

4.2.4.4 Collaboration

Figure 4.17 illustrates the interaction realizing DSMF processing between the components, described in the following.

- *Local Environment*
  A distributed hardware platform, e.g. vehicle-installed OBE, ECU, smartphone or the smart card of the interoperability use case. Runs the *HIRTE Component.*

- *Remote Environment*
  Either a hardware platform tightly integrated with the local module, e.g. a device with a smart card reader, another ECU in the same vehicle, or a center-side server. Runs the *Remote RTE* software.

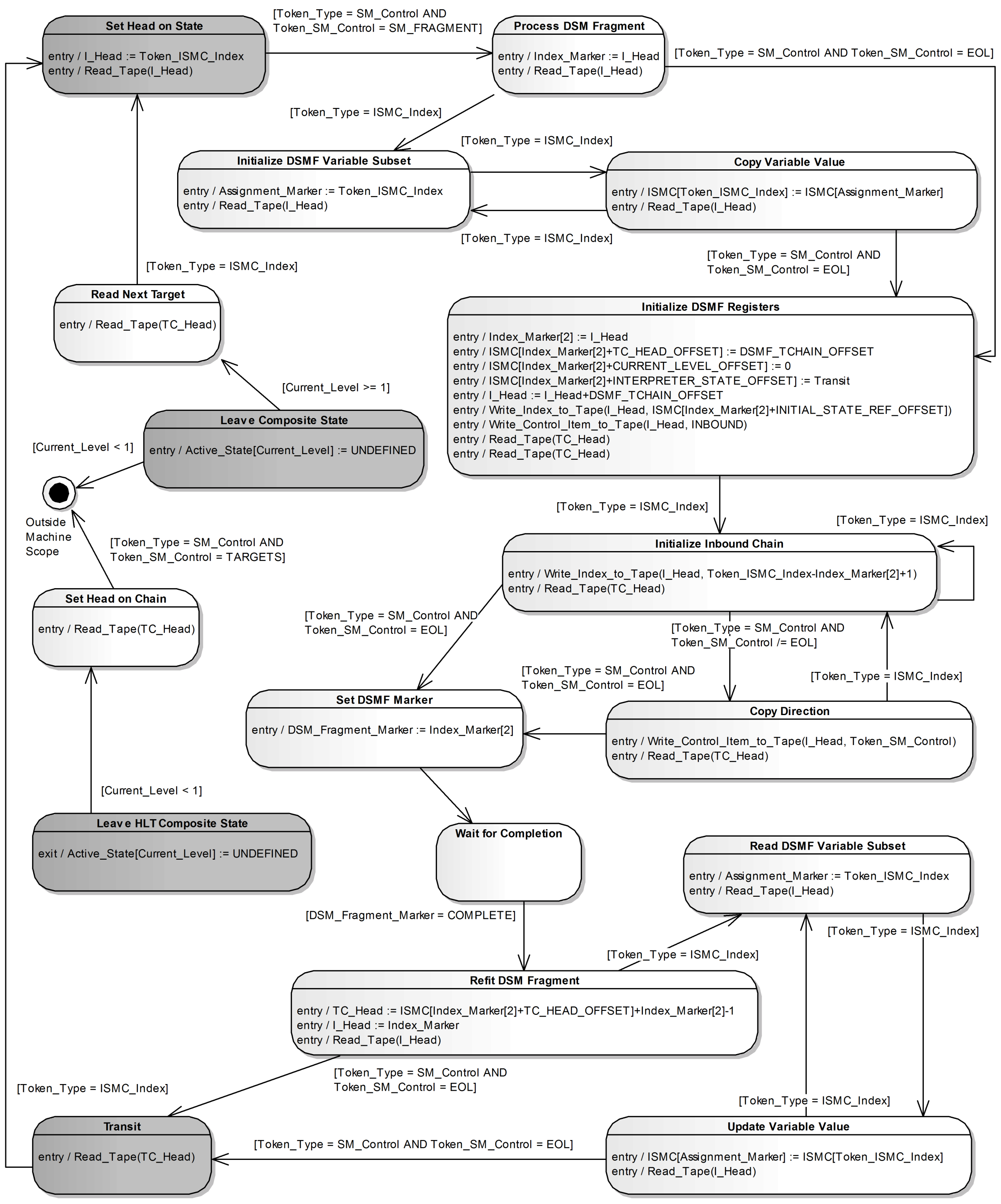

Set Head on State
entry / I_Head := Token_ISMC_Index
entry / Read_Tape(I_Head)
[Token_Type = SM_Control AND Token_SM_Control = SM_FRAGMENT]
Process DSM Fragment
entry / Index_Marker := I_Head
entry / Read_Tape(I_Head)
[Token_Type = SM_Control AND Token_SM_Control = EOL]
[Token_Type = ISMC_Index]
[Token_Type = ISMC_Index]
Initialize DSMF Variable Subset
entry / Assignment_Marker := Token_ISMC_Index
entry / Read_Tape(I_Head)
Copy Variable Value
entry / ISMC[Token_ISMC_Index] := ISMC[Assignment_Marker]
entry / Read_Tape(I_Head)
[Token_Type = ISMC_Index]
[Token_Type = SM_Control AND Token_SM_Control = EOL]
[Token_Type = ISMC_Index]
Read Next Target
entry / Read_Tape(TC_Head)
Initialize DSMF Registers
entry / Index_Marker[2] := I_Head
entry / ISMC[Index_Marker[2]+TC_HEAD_OFFSET] := DSMF_TCHAIN_OFFSET
entry / ISMC[Index_Marker[2]+CURRENT_LEVEL_OFFSET] := 0
entry / ISMC[Index_Marker[2]+INTERPRETER_STATE_OFFSET] := Transit
entry / I_Head := I_Head+DSMF_TCHAIN_OFFSET
entry / Write_Index_to_Tape(I_Head, ISMC[Index_Marker[2]+INITIAL_STATE_REF_OFFSET])
entry / Write_Control_Item_to_Tape(I_Head, INBOUND)
entry / Read_Tape(TC_Head)
entry / Read_Tape(TC_Head)
[Current_Level >= 1]
Leave Composite State
entry / Active_State[Current_Level] := UNDEFINED
[Current_Level < 1]
Outside Machine Scope
[Token_Type = ISMC_Index]
[Token_Type = ISMC_Index]
[Token_Type = SM_Control AND Token_SM_Control = TARGETS]
Initialize Inbound Chain
entry / Write_Index_to_Tape(I_Head, Token_ISMC_Index-Index_Marker[2]+1)
entry / Read_Tape(TC_Head)
Set Head on Chain
entry / Read_Tape(TC_Head)
[Token_Type = SM_Control AND Token_SM_Control = EOL]
[Token_Type = SM_Control AND Token_SM_Control /= EOL]
[Token_Type = SM_Control AND Token_SM_Control = EOL]
[Token_Type = ISMC_Index]
Set DSMF Marker
entry / DSM_Fragment_Marker := Index_Marker[2]
Copy Direction
entry / Write_Control_Item_to_Tape(I_Head, Token_SM_Control)
entry / Read_Tape(TC_Head)
[Current_Level < 1]
Leave HLT Composite State
exit / Active_State[Current_Level] := UNDEFINED
Wait for Completion
Read DSMF Variable Subset
entry / Assignment_Marker := Token_ISMC_Index
entry / Read_Tape(I_Head)
[DSM_Fragment_Marker = COMPLETE]
[Token_Type = ISMC_Index]
[Token_Type = ISMC_Index]
Refit DSM Fragment
entry / TC_Head := ISMC[Index_Marker[2]+TC_HEAD_OFFSET]+Index_Marker[2]-1
entry / I_Head := Index_Marker
entry / Read_Tape(I_Head)
[Token_Type = SM_Control AND Token_SM_Control = EOL]
[Token_Type = ISMC_Index]
[Token_Type = ISMC_Index]
Transit
entry / Read_Tape(TC_Head)
[Token_Type = SM_Control AND Token_SM_Control = EOL]
Update Variable Value
entry / ISMC[Assignment_Marker] := ISMC[Token_ISMC_Index]
entry / Read_Tape(I_Head)

Figure 4.15: ISMC Interpreter Automaton Extensions for DSMF

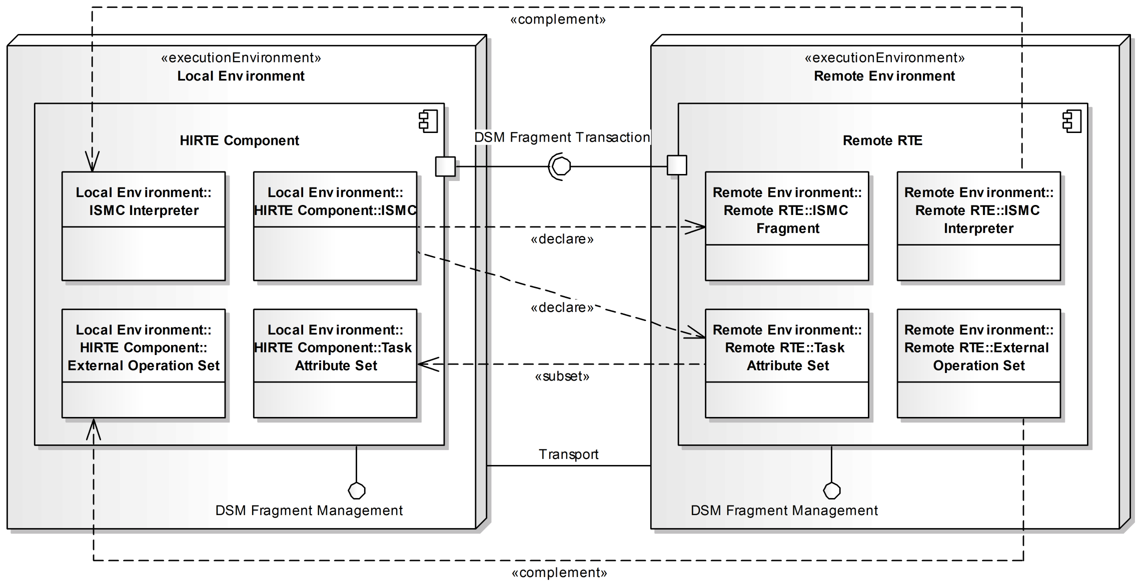


Figure 4.16: Distributable State Machine Fragment Pattern Structure

- *Transport Connection*
  Some physical link between the local and remote modules with low-level protocols up to the OSI transport layer. May range from an automotive bus wiring to a CN OTA connection.

- *DSM Fragment Management Interface*
  Requires the associated component to implement methods to *extract* a fragment byte subsequence from the *ISMC* on the local *Interpreter*'s *DSM Fragment Marker* referencing a DSMF, *encode* it as a message, or sequence of submessages in case of length restrictions, *decode* message(s) to ISMC and *replace* a fragment in the *ISMC* byte array. Additionally, this interface may include methods to *check requirements* (see previous section and below) of the fragment execution regarding operations to invoke and expected events/messages.

- *DSM Fragment Transaction Interface*
  For fragment distribution, the local *HIRTE Component* requires an interface that hides the specifics of the *Transport Connection* and provides methods to *send* and *receive* the ISMC subarrays encoded in a corresponding message format. This interface also has to regard application-specific addressing respectively miscellaneous message control issues.

- *HIRTE Component*
  Representing the software controlling processing of the *ISMC* that in turn

determines the behavior of the implemented service. The port with the associated (required) interface connection *DSM Fragment Transaction* is implemented by the input/output message queues as introduced in 4.2.3.4. Basically, the setup conforms to a VCU.

In a *Run* loop, the component continuously checks steps for the *ISMC Interpreter* reaching a distributable fragment (*DSM Fragment Marker*). On getting such reference, it reads out the corresponding byte string, determined by the starting index stored in the marker register plus the program length as stored in the fragment *ISMC Program Length* header entry (cmp. constraints in fig. 4.17). Depending on the *Transport Connection* and *Transaction* protocol transferable message size, the component *packs* the fragment into *n* message partitions. The encoded message is handed over to the local output queue buffer, and from there distributed to the remote input queue of the *Remote RTE*.

After remote processing, the updated message(s) are returned via the local input queue and subsequently *unpacked* to a decoded ISMC fragment. Its byte sequence is written to the original *ISMC*'s above index, replacing the fragment ISMC in a new state configuration. Prompting the interpreter to commence machine processing, the component sets the *DSM Fragment Marker* to *COMPLETE*.

- *Remote RTE*
  While designated RTE, this component generally represents an application that provides a RTE – interpreter, attributes and operations – for HIRTE ISMC and implements the corresponding interfaces for fragment transaction and management. It may also have other purposes beyond serving the HIRTE.

  The remote component receives the fragment as message(s) in its input queue. It *unpacks* the received data to one ISMC fragment byte sequence and uses the contained specified *requirements* to *check* for a valid executability in the given RTE: does the component support the required *operations* and *signals*, does it declare an *attribute set* matching the fragment's subset?

  If the capabilities are asserted, the fragment changes its role to a regular ISMC – *Set Tape* prepares the remote interpreter for processing of the machine program. Here, we assume that the remote *Set Tape* method strips or disregards the bytes preceding the ISMC specification, e.g. by resuming the head's position after reading out the requirements.

  The *remote execution loop* takes the same structure and flow as the established local *Run* processing. It *Step*s through the interpreter states, managing input/output message queues, invoking external operations where indicated and also distributing fragments, effectively enabling distribution cascades if an ISMC fragment contains another fragment. As the one distinction, a fragment program execution is expected to terminate with the state *Outside Machine Scope*, indicating a transition that left the fragment composite state and needs to continue in its composing machine's context, namely after returning the fragment to the local RTE.

Accordingly, on a proper termination, the processed fragment with updated attributes is repacked for transmission, the resulting message handed over to the remote component's output queue for passing back to the local input queue.

- *Local and Remote ISMC Interpreters*
  Process and execute an ISMC program conforming to their interpreter state automata. Local and remote instances are meant to be distinct insofar as the remote *complements* the local interpreter: depending on the capabilities – computing power, available resources, accessable devices, security levels, quality of service – the remote module provides native operations the local module cannot or is not allowed to realize.

- *Local and Remote External Operation Set*
  Provide procedures that elude implementation as native operations due to complexity or RTE-proprietary character (e.g. sensor access). Meaning of the *complements* dependency is equal to the relationship between local and remote interpreter, only referring to external operations. Summarized, this implies that the union of local and remote, native and external operation sets contains the set of operations invoked by the valid ISMC programs of a VCU system.

- *Local ISMC and Remote Fragment*
  Represent valid and executable (in relation to the given interpreter realizations) ISMC programs. With a composite state preceded by a *DSM FRAGMENT* opcode, the local *ISMC declares* the structure and content of the remote *ISMC Fragment*. Due to our implementation restrictions still in effect (recall 3.2.3), the differentiation of the two given elements in the pattern is intentional and significant: actual instances of both exist at any time, at least as containers, as it is not allowed to create the fragment dynamically.

  Likewise, the local *ISMC* declares the remote *Task Attribute Set* by specifying the *Variable Subset* mapping between original state machine and fragment variables: the remote *Task Attribute Set* is a subset of or equal to the set of locally mapped fragment variables.

- *Local and Remote Task Attribute Set*
  Compose and provide at least[17] all variables the ISMC and fragment external operations access, respectively. The *declares* dependency between

[17]Thus, the possible subset relation between mapped fragment variables and the remote

ISMC and remote attributes implies the *subset* dependency between the two attribute sets.

4.2.4.5 Implementation

The listings given in this section complete the sources of 4.2.3.5. First, listing 4.14 introduces the spec for DSMF processing.

Listing 4.14: ISMC Interpreter Spec Additions

```
package HIRTE_ISMC_Interpreter is
   ...

   ISMC_PROGRAM_LENGTH_OFFSET : constant Storage_Count := 78;
   DSM_FRAGMENT_MARKER_OFFSET : constant Storage_Count := 80;
   DSMF_TCHAIN_OFFSET : constant Storage_Count := 82;
   ...

   type ISMC_Automaton_State is ( ... Read_Next_Target, Process_DSM_Fragment,
      Initialize_DSMF_Variable_Subset, Copy_Variable_Value,
      Initialize_DSMF_Registers, Initialize_Inbound_Chain, Copy_Direction,
      Set_DSMF_Marker, Wait_for_Completion, Set_Head_on_Chain,
      Outside_Machine_Scope, Refit_DSM_Fragment, Read_DSMF_Variable_Subset,
      Update_Variable_Value);

   for ISMC_Automaton_State use ( ... Read_Next_Target => 71,
      Process_DSM_Fragment => 72, Initialize_DSMF_Variable_Subset => 73,
      Copy_Variable_Value => 74,  Initialize_DSMF_Registers => 75,
      Initialize_Inbound_Chain => 76, Copy_Direction => 77,
      Set_DSMF_Marker => 78, Wait_for_Completion => 79,
      Set_Head_on_Chain => 80, Outside_Machine_Scope => 81,
      Refit_DSM_Fragment => 82, Read_DSMF_Variable_Subset => 83,
      Update_Variable_Value => 84);
   ...

   type ISMC_Control_Item is (... DSM_FRAGMENT);
   for ISMC_Control_Item use (... DSM_FRAGMENT => 22);
   ...

   protected type ISMC_Automaton is
      ...

      procedure Set_DSM_Fragment_Marker(I : in ISMC_Index);
      function Get_DSM_Fragment_Marker return ISMC_Index;
      procedure Set_ISMC_Program_Length(I : in ISMC_Index);
      function Get_ISMC_Program_Length return ISMC_Index;
      ...

   end ISMC_Automaton;
```

The spec adds attributes and opcode of the syntax definition of listing 4.13 as well as the states of the automaton of fig. 4.15. To confer the conceptual

attribute set: strictly speaking, the attribute set record is necessary only for the external operations as a convenient way to access the corresponding byte sequence in the ISMC (compare its record representation). Native operations directly address this sequence without a detour over a declared attribute set record and in consequence do not need explicit variable declarations in any record.

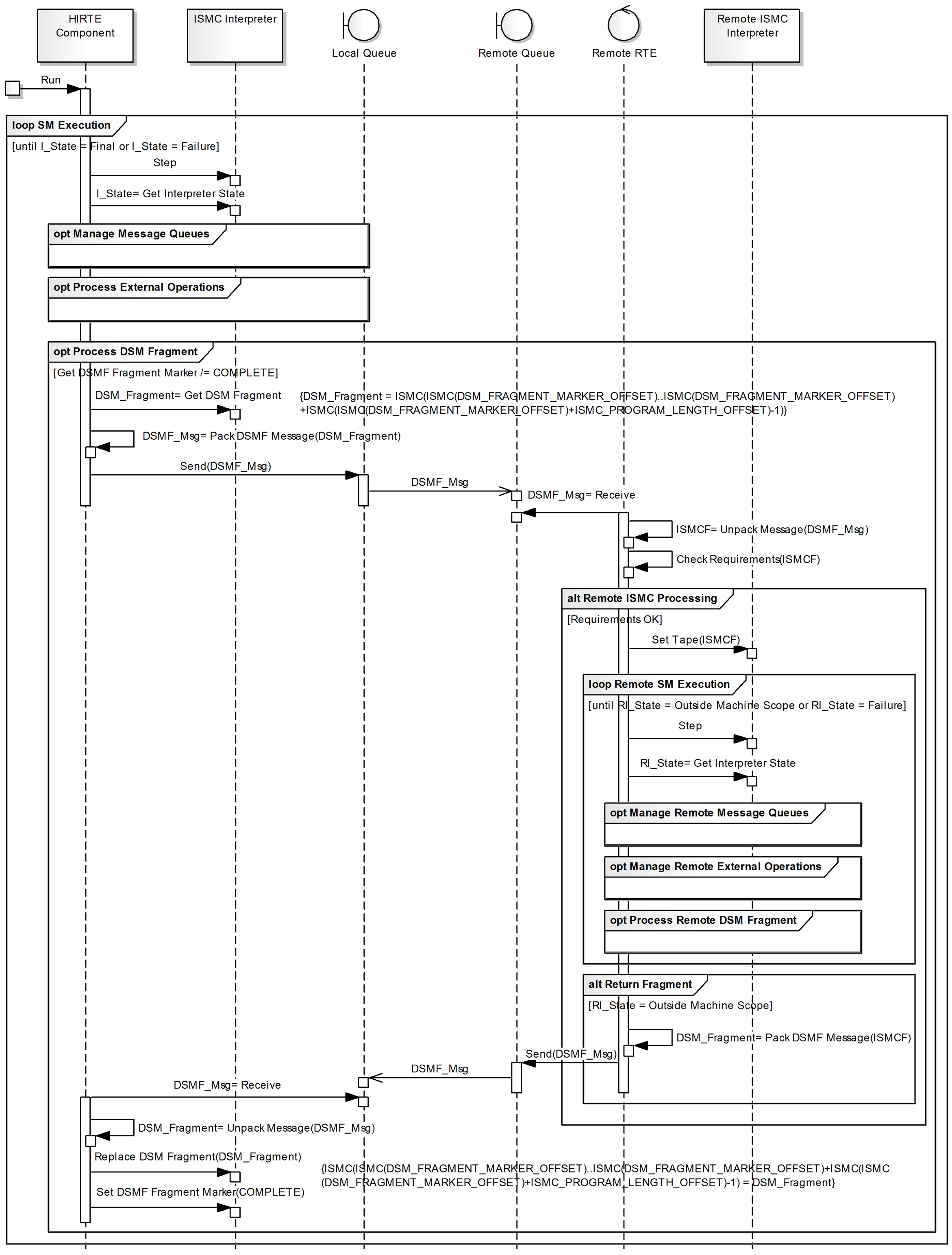

HIRTE Component
ISMC Interpreter
Local Queue
Remote Queue
Remote RTE
Remote ISMC Interpreter
Run
loop SM Execution
[until I_State = Final or I_State = Failure]
Step
I_State= Get Interpreter State
opt Manage Message Queues
opt Process External Operations
opt Process DSM Fragment
[Get DSMF Fragment Marker /= COMPLETE]
DSM_Fragment= Get DSM Fragment
{DSM_Fragment = ISMC(ISMC(DSM_FRAGMENT_MARKER_OFFSET)..ISMC(DSM_FRAGMENT_MARKER_OFFSET)
+ISMC(ISMC(DSM_FRAGMENT_MARKER_OFFSET)+ISMC_PROGRAM_LENGTH_OFFSET)-1)}
DSMF_Msg= Pack DSMF Message(DSM_Fragment)
Send(DSMF_Msg)
DSMF_Msg
DSMF_Msg= Receive
ISMCF= Unpack Message(DSMF_Msg)
Check Requirements(ISMCF)
alt Remote ISMC Processing
[Requirements OK]
Set Tape(ISMCF)
loop Remote SM Execution
[until RI_State = Outside Machine Scope or RI_State = Failure]
Step
RI_State= Get Interpreter State
opt Manage Remote Message Queues
opt Manage Remote External Operations
opt Process Remote DSM Fragment
alt Return Fragment
[RI_State = Outside Machine Scope]
DSM_Fragment= Pack DSMF Message(ISMCF)
Send(DSMF_Msg)
DSMF_Msg
DSMF_Msg= Receive
DSM_Fragment= Unpack Message(DSMF_Msg)
Replace DSM Fragment(DSM_Fragment)
{ISMC(ISMC(DSM_FRAGMENT_MARKER_OFFSET)..ISMC(DSM_FRAGMENT_MARKER_OFFSET)+ISMC(ISMC
(DSM_FRAGMENT_MARKER_OFFSET)+ISMC_PROGRAM_LENGTH_OFFSET)-1) = DSM_Fragment}
Set DSMF Fragment Marker(COMPLETE)

Figure 4.17: DSMF Processing Interaction between Local and Remote RTE

orthogonality of ISMC and fragment on the implementation, we do not provide a specialized interpreter for DSMFs, instead integrating the mechanism directly to avoid module variants. Accordingly, listing 4.15 realizes this specification.

Listing 4.15: ISMC Interpreter Body Additions

```
package body HIRTE_ISMC_Interpreter is

   protected body ISMC_Automaton is
   ...

   procedure Step is
   begin
      ...

            when Read_Next_Target =>

               Read_Tape(ISMC(TC_HEAD_OFFSET..TC_HEAD_OFFSET+1));

               if Token_Type = T_ISMC_Index then
                  Interpreter_State := Set_Head_on_State;
               else
                  Interpreter_State := Failure;
               end if;

            when Process_DSM_Fragment =>

               ISMC(INDEX_MARKER_OFFSET..INDEX_MARKER_OFFSET+1) :=
                  ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1);
               Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));

               if Token_Type = T_ISMC_Index then
                  Interpreter_State := Initialize_DSMF_Variable_Subset;
               else
                  if Token_Type = T_SM_Control and Token_SM_Control = EOL then
                     Interpreter_State := Initialize_DSMF_Registers;
                  else
                     Interpreter_State := Failure;
                  end if;
               end if;

            when Initialize_DSMF_Variable_Subset =>

               ISMC(ASSIGNMENT_MARKER_OFFSET..ASSIGNMENT_MARKER_OFFSET+1) :=
                  Write_ISMC_Index(Token_ISMC_Index);
               Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));

               if Token_Type = T_ISMC_Index then
                  Interpreter_State := Copy_Variable_Value;
               else
                  Interpreter_State := Failure;
               end if;

            when Copy_Variable_Value =>

               -- Prototype copies four bytes as integer:
               ISMC(Storage_Count(Token_ISMC_Index)
                  ..Storage_Count(Token_ISMC_Index)+3) :=
                  ISMC(Storage_Count(Read_ISMC_Index(ISMC(ASSIGNMENT_MARKER_OFFSET
                     ..ASSIGNMENT_MARKER_OFFSET+1)))
                        ..Storage_Count(Read_ISMC_Index(ISMC(
                           ASSIGNMENT_MARKER_OFFSET..ASSIGNMENT_MARKER_OFFSET+1)))+3);
```

```
         Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));

         if Token_Type = T_ISMC_Index then
            Interpreter_State := Initialize_DSMF_Variable_Subset;
         else
            if Token_Type = T_SM_Control and Token_SM_Control = EOL then
               Interpreter_State := Initialize_DSMF_Registers;
            else
               Interpreter_State := Failure;
            end if;
         end if;

      when Initialize_DSMF_Registers =>

         ISMC(INDEX_MARKER_OFFSET+2..INDEX_MARKER_OFFSET+3) :=
            ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1);
         ISMC(Storage_Offset(Read_ISMC_Index(ISMC(INDEX_MARKER_OFFSET+2
            ..INDEX_MARKER_OFFSET+3)))+TC_HEAD_OFFSET-1
               ..Storage_Offset(Read_ISMC_Index(ISMC(INDEX_MARKER_OFFSET+2
                  ..INDEX_MARKER_OFFSET+3)))+TC_HEAD_OFFSET) :=
            Write_ISMC_Index(ISMC_Index(DSMF_TCHAIN_OFFSET));
         ISMC(Storage_Offset(Read_ISMC_Index(ISMC(INDEX_MARKER_OFFSET+2
            ..INDEX_MARKER_OFFSET+3)))+CURRENT_LEVEL_OFFSET-1) := 0;
         ISMC(Storage_Offset(Read_ISMC_Index(ISMC(INDEX_MARKER_OFFSET+2
            ..INDEX_MARKER_OFFSET+3)))+INTERPRETER_STATE_OFFSET-1) :=
            Write_ISMC_Automaton_State(Transit);
         ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1) :=
            Write_ISMC_Index(Read_ISMC_Index(ISMC(I_HEAD_OFFSET
               ..I_HEAD_OFFSET+1))+ISMC_Index(DSMF_TCHAIN_OFFSET-1));
         -- Per default set the fragment's CS to the first state to enter
         -- remotely:
         Write_Index_to_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1),
            Read_ISMC_Index(ISMC(Storage_Offset(
               Read_ISMC_Index(ISMC(INDEX_MARKER_OFFSET+2
                  ..INDEX_MARKER_OFFSET+3)))+INITIAL_STATE_REF_OFFSET-1
                    ..Storage_Offset(Read_ISMC_Index(ISMC(INDEX_MARKER_OFFSET+2
                        ..INDEX_MARKER_OFFSET+3)))+INITIAL_STATE_REF_OFFSET))));
         Write_Control_Item_to_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1),
            INBOUND);
         Read_Tape(ISMC(TC_HEAD_OFFSET..TC_HEAD_OFFSET+1)); -- Discard inbound
                                                            -- flag
         Read_Tape(ISMC(TC_HEAD_OFFSET..TC_HEAD_OFFSET+1));

         if Token_Type = T_ISMC_Index then
            Interpreter_State := Initialize_Inbound_Chain;
         else
            Interpreter_State := Failure;
         end if;

      when Initialize_Inbound_Chain =>

         Write_Index_to_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1),
            Token_ISMC_Index-Read_ISMC_Index(ISMC(INDEX_MARKER_OFFSET+2
               ..INDEX_MARKER_OFFSET+3))+1);
         Read_Tape(ISMC(TC_HEAD_OFFSET..TC_HEAD_OFFSET+1));

         if Token_Type = T_ISMC_Index then
            Interpreter_State := Initialize_Inbound_Chain;
         else
            if Token_Type = T_SM_Control then
               if Token_SM_Control = EOL then
                  Interpreter_State := Set_DSMF_Marker;
         else
            Interpreter_State := Copy_Direction;
         end if;
      else
         Interpreter_State := Failure;
      end if;
   end if;

when Copy_Direction =>

   Write_Control_Item_to_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1),
      Token_SM_Control);
   Read_Tape(ISMC(TC_HEAD_OFFSET..TC_HEAD_OFFSET+1));

   if Token_Type = T_ISMC_Index then
      Interpreter_State := Initialize_Inbound_Chain;
   else
      if Token_Type = T_SM_Control and Token_SM_Control = EOL then
         Interpreter_State := Set_DSMF_Marker;
      else
         Interpreter_State := Failure;
      end if;
   end if;

when Set_DSMF_Marker =>

   ISMC(DSM_FRAGMENT_MARKER_OFFSET..DSM_FRAGMENT_MARKER_OFFSET+1) :=
      ISMC(INDEX_MARKER_OFFSET+2..INDEX_MARKER_OFFSET+3);

   Interpreter_State := Wait_for_Completion;

when Wait_for_Completion =>

   if Read_ISMC_Index(ISMC(DSM_FRAGMENT_MARKER_OFFSET
      ..DSM_FRAGMENT_MARKER_OFFSET+1)) = COMPLETE then
      Interpreter_State := Refit_DSM_Fragment;
   end if;

when Refit_DSM_Fragment =>

   ISMC(TC_HEAD_OFFSET..TC_HEAD_OFFSET+1) :=
      ISMC(Storage_Offset(Read_ISMC_Index(ISMC(
         INDEX_MARKER_OFFSET+2..INDEX_MARKER_OFFSET+3)))+TC_HEAD_OFFSET−1
            ..Storage_Offset(Read_ISMC_Index(ISMC(INDEX_MARKER_OFFSET+2
               ..INDEX_MARKER_OFFSET+3)))+TC_HEAD_OFFSET);
   ISMC(TC_HEAD_OFFSET..TC_HEAD_OFFSET+1) :=
      Write_ISMC_Index(Read_ISMC_Index(
         ISMC(TC_HEAD_OFFSET..TC_HEAD_OFFSET+1))
            +Read_ISMC_Index(ISMC(INDEX_MARKER_OFFSET+2
               ..INDEX_MARKER_OFFSET+3))−1);
   ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1) :=
      ISMC(INDEX_MARKER_OFFSET..INDEX_MARKER_OFFSET+1);
   Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));

   if Token_Type = T_ISMC_Index then
      Interpreter_State := Read_DSMF_Variable_Subset;
   else
      if Token_Type = T_SM_Control and Token_SM_Control = EOL then
         Interpreter_State := Transit;
      else
         Interpreter_State := Failure;
      end if;
```

```
               end if;

            when Read_DSMF_Variable_Subset =>

               ISMC(ASSIGNMENT_MARKER_OFFSET..ASSIGNMENT_MARKER_OFFSET+1) :=
                  Write_ISMC_Index(Token_ISMC_Index);
               Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));

               if Token_Type = T_ISMC_Index then
                  Interpreter_State := Update_Variable_Value;
               else
                  Interpreter_State := Failure;
               end if;

            when Update_Variable_Value =>

               -- Prototype copies four bytes as integer:
               ISMC(Storage_Count(Read_ISMC_Index(ISMC(
                  ASSIGNMENT_MARKER_OFFSET..ASSIGNMENT_MARKER_OFFSET+1)))
                     ..Storage_Count(Read_ISMC_Index(ISMC(ASSIGNMENT_MARKER_OFFSET
                        ..ASSIGNMENT_MARKER_OFFSET+1)))+3) :=
                  ISMC(Storage_Count(Token_ISMC_Index)
                     ..Storage_Count(Token_ISMC_Index)+3);
               Read_Tape(ISMC(I_HEAD_OFFSET..I_HEAD_OFFSET+1));

               if Token_Type = T_ISMC_Index then
                  Interpreter_State := Read_DSMF_Variable_Subset;
               else
                  if Token_Type = T_SM_Control and Token_SM_Control = EOL then
                     Interpreter_State := Transit;
                  else
                     Interpreter_State := Failure;
                  end if;
               end if;

            when Set_Head_on_Chain =>

               Read_Tape(ISMC(TC_HEAD_OFFSET..TC_HEAD_OFFSET+1));

               if Token_Type = T_SM_Control and Token_SM_Control = TARGETS then
                  Interpreter_State := Outside_Machine_Scope;
               else
                  Interpreter_State := Failure;
               end if;

            when Outside_Machine_Scope =>

               null;
            ...

      end;

   end ISMC_Automaton;

end HIRTE_ISMC_Interpreter;
```

For the sake of completeness, listing 4.16 shows a plain implementation of the *Run* procedure. Its external operations refer to the state machine example of 4.2.1.5.

Listing 4.16: VCU Component Run Procedure Example

```
procedure Run is

   I_State : ISMC_Automaton_State;
   Op_ID : External_Operation_Set.External_Operation_ID;
   Last_Message_Body : Messages.Binary_Buffer;
   OK : Boolean := False;

begin

   loop -- Main execution loop

      -- Process an interpreter step ...
      Machine.Step;
      -- ... and get the resulting state:
      I_State := Machine.Get_Interpreter_State;

      -- If the interpreter message buffer is empty ...
      if Machine.Get_Last_Message_Buffer = UNDEFINED_MSG then
         -- ... check for a new message ...
         Message_Queues.Get_Msg(From_Kernel_Queue.all, Last_Message_Body, OK);
         if OK then
            -- ... and hand it over to the interpreter machine:
            Machine.Set_Last_Message_Buffer(Last_Message_Body);
         end if;
      end if;

      -- If the interpreter requests execution of an external operation ...
      if Machine.Get_External_Operation_Marker /= COMPLETE then
         -- ... retrieve its identification ...
         Op_ID := Read_External_Operation_ID(Machine.Get_Tape(
            Storage_Offset(Machine.Get_External_Operation_Marker)
               ..Storage_Offset(Machine.Get_External_Operation_Marker)+1));

         -- ... and call it:
         case Op_ID is

            when External_Operation_Set.A_Simple_1_Entry_ID =>
               External_Operation_Set.A_Simple_1_Entry;

            when External_Operation_Set.A_Simple_1_Exit =>
               External_Operation_Set.A_Simple_1_Exit;

            when External_Operation_Set.A_privateOp1 =>
               External_Operation_Set.A_privateOp1;

            when External_Operation_Set.Composite_1_Exit =>
               External_Operation_Set.Composite_1_Exit;

            when External_Operation_Set.Guard_Composite_1_TO_Simple_3 =>
               External_Operation_Set.Guard_Composite_1_TO_Simple_3;

            when External_Operation_Set.Guard_Composite_2_TO_Simple_3 =>
               External_Operation_Set.Guard_Composite_2_TO_Simple_3;

         end case;

         -- On return, notify the interpreter:
         Machine.Set_External_Operation_Marker(COMPLETE);

      end if;

```

```
        -- Stop or interrupt processing on finishing the ISMC processing
        -- or encountering a fatal problem ...
        exit when I_State = Final or I_State = Failure
           -- ... or leaving the ISMC state hierarchy ...
           or I_State = Outside_Machine_Scope
           -- ... or to distribute a fragment for remote execution:
           or Machine.Get_DSM_Fragment_Marker /= COMPLETE;
           -- Variation point: the VCU might directly pack the fragment as
           -- a message and distribute it via an output queue.

    end loop;

end;
```

4.2.4.6 Variants and Extensions

fragment aspects

The concept of DSMFs is not necessarily tied exclusively to ISMC. If we abstract the essential aspects of the fragments, we find *segregation* and *handover*. The former takes a static view and refers to the systematic practical pursuit of *self-containment* (cmp. 4.1.3) in states: the automaton structures need to allow partitioning under the constraint of maintained integrity and validity of all (sub-)machine attributes relevant for processing. The latter, a dynamic view, pertains to "hotgrafting" an automaton part, i.e. transfering at run-time, while ensuring invariant coherence and consistency of its static structures and dynamic traces – at any step, independent from the local fragment distribution, the exhibited behavior has to conform to the original automaton.

fragmenting SRSM

With these aspects identified, we revisit the SRSM pattern of 4.2.1. A fundamental prerequisite for the segregation of a fragment composite state is a serviceable state hierarchy that makes the composition of states explicit. ISMC achieves this fragment cohesion by conforming to the syntax of listing 4.13; per definitionem, its states occupy a coherent byte sequence. In contrast, a SRSM structure specializes the hierarchical relation on its utilizing element: it is implied in the transition chains. Consequently, compilation of the fragment state set demands deriving the compositional information from these records. As the conceptual structure and underlying processing approaches of SRSMs and ISMC are equivalent (as established in 4.2.3.2), we can reduce the handover of SRSM fragments to the questions of serialization and selective processing. Serializing the state and transition records does not involve high efforts if we substitute pointers with indices (sketched in 4.2.1.6). Processing the structures pre and post handover requires limiting the traversion of transition chains to the states contained in the fragment while preserving the remaining order for the replacement. This merely requires additional (boolean) discriminants and corresponding exit conditions in the *Interprete* and *Run Machine* procedures.

parametrization

The fragment distribution approach – handing over executable program parts to remote components – unlocks another avenue: *parametrized fragments*. While still preserving the in- and outgoing transitions, we can introduce dynamic[18] composition respectively compilation of the fragment programs. The ETC domain suggests a number of use cases, especially in the context of toll atom identification and rating to charging (cmp. 2.2.1). To generate receipt records, e.g. to answer automatic enforcement challenges, the local HIRTE component may compile specific tariff calculation instruction machines for an infrastructure access (segment, tour, area). This customized automaton may then be passed to a remote component for computation without a complex tariff scheme required externally. A similar process is feasible for traffic infrastructure geometry identification. Computation instructions for a specific infrastructure may be passed to a component outfitted with corresponding sensors; again, this external component does not need access to the tolling service's operating database. The results are returned encoded in the updated ISMC, its proper transitions having fired to be concluded locally, depending on the result. Answering the question for a robust, safe formalism to distribute tariff calculation instructions is crucial for the EETS scenario ([EU09]).

4.3 Chapter Conclusion

why the detour

It would seem that the results of chapters 2 and 3 presented us with an objective and set of requirements sufficient to design a proper ETC application architecture. The aspects, business processes, interfaces and logics of the domain introduction served as alignment of the software's modular setup: without needing the details of infrastructure identification, communication protocols or payment record structures, it already gave us a clear idea of the separation of concerns in the sense of layers, basic and composite services that will in concert reflect the aspects and realize the application. The substantiations of reliability and economy, their set of concrete requirements and regulations even instructed us how exactly to implement these ideas.

wider benefits

In this light, the results of this chapter might appear as a kind of detour. Instead of describing the actual solution, it presents a set of patterns; generic approaches, that is. This is owed to the fact that we introduced the domain of ETC as one representative of a genuine, mission-critical type of telematics system. This example allowed us to reason purposefully about the characteristics of distributed, embedded software with various and variable environment interactions. Consequently, the intermediate step of the patterns offers building blocks not only for ETC deployments, but for any telematics system with similar requirements – found in such dissimilar subdomains as healthcare and defense.

some stereotypes and the self-contained state

Earlier (3.3.1), we selected the UML as modeling language of choice to illustrate the results of this work. For the description of the patterns, the first section of this chapter derived a number of supplementary stereotypes –

[18]Only related to the ISMC *content*, not structure – the static programming techniques stay in effect.

behavioral with *procedure*, *function* and *task*, *data structure* with *static memory segment*, *cyclic buffer* and *record* – from the UML metamodel. A more significant stereotype was introduced as the *distributable state machine fragment*. With the notion of a *self-contained state*, it extends the concept of the state automaton by a selectively transferable part: a substate that entrains and aggregates all information to be meaningful, or interpretable, beyond the context of its original state machine.

the HIRTE core patterns

The *component* is a basic element of our architecture. It is supposed to be a module realizing some service with defined input/output interfaces in the form of message queues. Our component's behavior is determined by a state automaton. With the preparatory work, the chapter introduced the elementary design patterns and structures that make up the HIRTE core:

Statically Resolvable State Machine Pattern – a blueprint for the implementation of a component conforming to a hierarchical state machine. Superficially, it exhibits a similar structure and behavior as established solutions (a state automaton per se is nothing original, after all). Internally, the pattern dismisses all elements and configurations that might rely on dynamic memory allocation, polymorphy, dispatching and similar mechanisms considered unsafe in a high-integrity deployment. In contrast to existing works, despite this robustness it still maintains a sufficient expressiveness to match the complexity of the services of the given domain.

While not discouraging manual programming, the setup of the pattern generally aims at automatic generation of sources, i.e. model transformation. This is illustrated by the instantiation example, which follows a systematic and repetitive program composition.

State Tracer Pattern – structures and refines the utilization of a dedicated, statically allocated memory segment to log the system state configuration and its traces in (near) real-time. To this end, it avoids access collisions and mutual exclusion.

Virtual Control Unit Pattern – evolves the concept of a component implemented *as* a state machine to a scalable, high-integrity RTE *for* state automaton programs. The core VCU implementation is lean and extendable, consisting of a program interpreter, message queues, a set of procedures complementing the interpreter features and a set of variables for these operations. In the process, the pattern introduces the automaton programs in the form of *Interpretable State Machine Code* (ISMC), which realize the respective service, or actual application, of the VCU component.

Instead of the common deployments of virtual machines – either providing the extensive functionality of a full-scale OO language RTE or emulating a whole computer for an OS –, the VCU approach establishes networks of distributed lightweight modules controlled by restrictive programs. This follows the idea of reducing risks and supporting robustness of the overall system by reducing the complexity of the single element, working toward the notion of *composability* as defined in [Ko97].

With regard to the general approach of virtualization, we effectively virtualize a conceptual ECU (whereas an abstract one instead of some specific type); a small, efficiently controllable element. Correspondingly, a set of VCUs is meant to run on a single processor, be distributed over a network of processors (cmp. e.g. the automotive integration proposal of 3.1.3.2), but the case of one VCU spanning more than one processor is no relevant scenario of the given – embedded systems – domain.

Distributable State Machine Fragment Pattern – applies the *self-contained state* to VCUs and ISMC. An additional opcode allows fragment declarations of ISMC composite states. Transitions entering a fragment prompt the VCU to extract the corresponding byte segment, including all variables accessed by this subprogram, and transfer it to a remote RTE for processing. On an outgoing transition, the fragment is returned to the local VCU, replaced in the ISMC and the transition resumed.

This conceptual extension enables a VCU network to distribute its activities over a variety of RTEs, with the prerequisites of a remote ISMC interpreter and supported fragment transaction protocols. Among the use cases, we find *smart card interoperability* (cmp. 3.1.3.3; relevant for the economy of our solution) and the general ability to safely delegate computations to specialized components – regarding security, sensors, computation power, communication devices.

reference sources versus abandoned alternatives

All pattern descriptions feature a reference source in Ada, with the restricting *pragma*s in effect. Besides the purposes of standard implementations that an actual system may adopt – directly, refactored or ported to another language –, and proof-of-concept prototypes during the elaboration of this work, the Ada specs and bodies effectively serve as operational semantics of the pattern models. Here, and especially with the presentation of the ISMC semantics, we consciously infringe on the all-to-common norm of algebraic specification of operational semantics, prominently in the form of *Structured Operational Semantics* (SOS, [Pl04]). Under the pretense of scientific universality, too many publications thus *a)* (semantically) stay on a level of arbitrariness when they claim to present a concrete solution and *b)* invite fundamental mistakes by forgoing the relentlessnes of automated checks of their results, e.g. by a compiler. Our experience with above *a*-cases showed us that applying a comprehensive SOS to an actual system is not "trivial" (like often postulated in these works), and found results of works that were almost rendered completely useless because of *b*-cases[19].

requirements revisited

Even with the HIRTE application architecture only due next chapter, it is already worthwhile to revisit the seven requirements. Instead of examining one integrated architectural solution for fulfillment later, we can match each building block to the requirement set, allowing a more selective composition of a solution regarding the aspects touched.

Requirement I (centralized control) – is the one requirement we so far only served implicitly and will answer in the next chapter 5. We did not (yet) specify a controlling instance, except incidentally in the State Tracer Pattern. Nevertheless, this chapter provides the proper tools for the job. Stepwise execution of the VCUs, the immediate availability of the states in the State Tracer and defined interfaces to both SRSM and VCU components are sine qua non, necessary means to orchestrate the elements.

Requirement II (system state automata) – designates the underlying structural and behavioral principle of all patterns. Note that the patterns not only retain HIRTE-constituted application components to adhere to the state automaton alignment, but also consistently apply this design to their interpreter infrastructure. This facilitates methodical validation and eventual modification of a HIRTE solution by ensuring a formal, uniform system structure.

Requirement III (transparency of activities over time) – is explicitly fulfilled by the State Tracer, which provides the components (including a centralized control instance) with a safe interface to immediately log (and monitor) their behavior. The thusly emerging traces always provide a clear picture of the incorporated modules' activities.

Requirement IV (component organization) – is met by the SRSM and VCU patterns, which both employ the *HIRTE Component* with its message queues to compose the structures implementing the state automaton behavior, i.e. some service state machine conforming to II is always encapsuled and represented by a component.

Requirement V (statically determinable implementation) – is enforced by the profile and restriction pragmas imposed on the source code. Hence, the pattern design considers the implied constraints for the model, only admitting statically implementable structures.

Requirement VI (deterministic run-time behavior) – is answered indirectly; it concerns the scheduling of a set of components and message

[19]This is by no means a statement against the sophistication and usefulness of the SOS approach in general.

queue organization. We did not specify a scheduler or queue management, and will not in detail, as valid solutions for high-integrity applications exist. Essential for the HIRTE patterns is an effective option of controllability. That is granted by the component *Run* and *Step* methods, which may apply suitable strategies to the machine respectively interpreter invocations (preferably *fixed priority scheduling*, like *rate monotonic*; cmp. [BW09], [Bu05]) and queue access (e.g. FIFO without requeuing). Additionally, the pattern decomposition (component logics, queues, operation sets) allows for efficient analysis of determining factors, e.g. run-time of native and external operations.

Requirement VII (persistency of activities) – reflects in the ISMC approach: in a VCU-exclusive setup, the corresponding set of binary strings represents a complete system state. In conjunction with interpreter *Step*s, ISMC permits transaction based execution, e.g. by keeping persistent copies of the byte sequences that are updated and commited only after faultless step processing. While certain real-time dependencies of the activities (cue sensor readouts) still need consideration, the software can perpetuate a consistent state configuration – particularly crucial as well as effective for payment and operating data transactions.

Now that this chapter produced requirements-conformant building blocks for high-integrity telematics architectures, the next chapter may put them to – ETC domain-specific – use.

Five

HIRTE Application Architecture Patterns

... in which we finally illustrate how to build a software solution in conformance to our requirements of reliability and economy. Again, a set of patterns achieves this: they define the three successive views on a HIRTE architecture – the *framework*, the *implementation* and *application*. To come full circle from our introduction, the latter pattern sketches an ETC setup built of HIRTE components.

The architecture elements introduced in the previous chapter constitute a complex infrastructure for applications. Isolating each partial solution in a pattern ensures broader applicability in the general telematics domain. Any high-integrity system may select a proper subset of the offered elements for realization of its specific use cases.

returning to ETC

Nevertheless, we chose a specimen of telematics systems as a representative for this work. Consequently, this chapter will gradually refocus our endeavors on ETC to illustrate the methodical approach of a HIRTE implementation.

An actual deployment of an ETC system based on the HIRTE architecture has to be able to reflect the aspects introduced in 2.3. While the software processes determine hardware attributes like memory size, processor capabilities and communication service bandwidth, the distribution of the software modules over a technological infrastructure is usually the result of a business case. In consequence, an ETC reference architecture has to be flexible: portable, modular and scaleable. Here however, this flexibility has to effect a more abstract level than the run-time implementation, generally enabling systematic shifting and changing of interfaces, selective exchange of components (in development as well as during operation) and easy (economical) introduction of additional vehicle or infrastructure types.

5.1 HIRTE Framework Pattern

5.1.1 Abstract

The building blocks of the previous chapter each cover a defined aspect of the HIRTE architectural concept. In order to present a coherent foundation for applications, we need to establish the relations between these core pattern elements, composing a *framework* structure. To that end, packages express the separation of concerns between the contained components. Regarding an actual application, this still illustrates a meta-level: a service infrastructure will rely on the package contents as development reference to realize their structures and processes.

While it might appear redundant to specify a pattern – which is generic in itself – for a framework, commonly already representing a generic software structure, this pattern also lays out details of application-specific parts. Thus, it goes beyond e.g. a mere API structure, strictly only later integrated by application modules, instead conveying a sensible arrangement for selective and delimited specialization for an actual mission-critical telematics respectively ETC deployment. Also, an actual framework as manifestation of the pattern will provide concrete action sets, messages and queues.

A general consideration of the framework notion regarding the HIRTE is given in 6.2, after the description of all three application architecture patterns.

5.1.2 Structure

Fig. 5.1 describes the pattern's structure.

5.1.3 Collaboration

- *Generic High Integrity Run-time Environment*
  Contains components for the interpretation and controlled execution of state machine structures, thusly providing the core of the HIRTE. The package is denominated *generic*[1], as it is not concerned with functionality beyond the run-time processing of the machine states and their basic native actions.

  At this point, we introduce a *Kernel*, the control component as signified in 4.2.2, into HIRTE: it assumes the role of a central, integrating hub for all infrastructure elements of the system. While certain characteristics resemble that of a middleware, we use the term "kernel" for this control module of HIRTE. First of all, a middleware often is used to encapsule (or hide) the implementation of communications and connections between modules or layers of a system. In contrast, the HIRTE kernel controls and

[1]"Generic" still inside the HIRTE domain; not transgressing the overall domain-specific approach.

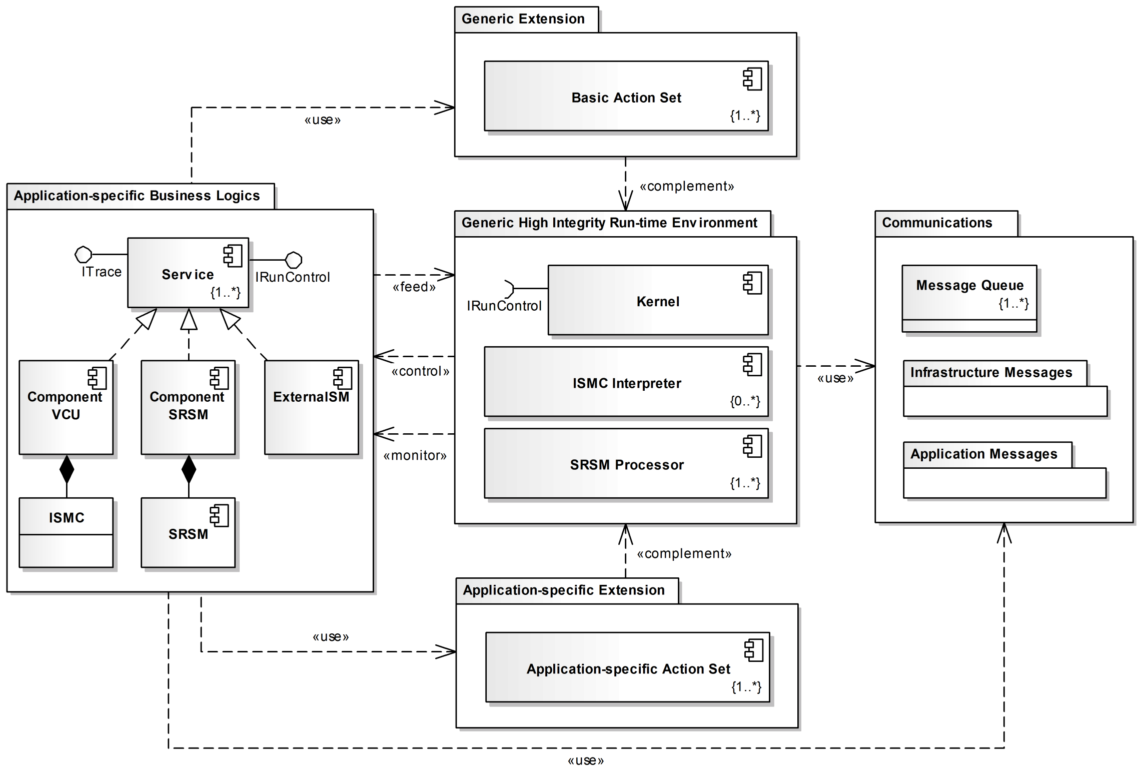


Figure 5.1: HIRTE Framework Pattern Structure

monitors the module transactions, while explicitly exposing the behavior of all elements (step semantics, state tracer). Secondly, a middleware is usually interpreted as a kind of mediating service layer. Our kernel on the other hand may implement elementary services for the components, effectively enabling it to replace the operating system for these.

Employing the *IRunControl* interface, it *controls* the services of an application. *IRunControl* defines the operations *Run* (execute a machine until it reaches *Final*, *Failure* or *Outside_ Machine_ Scope* states), *Step* (process a machine step as defined for SRSMs or ISMC) and *SetTape* (in the case of an ISMC component) as described in 4.2.1 and 4.2.3. Additionally, a *State Tracer* memory sequence as part of the kernel instance retains *monitoring* data, collated by the mechanism of 4.2.2.

For both interactions with the components and for controlling interactions between components, the kernel *uses* a *Communications* package.

The component *ISMC Interpreter* contains the spec and implementation of the module of the same name, and the record specification of an ISMC program, representing the generic SM concerns of the VCU (cmp. fig.

4.14). Correspondingly, *SRSM Processor* aggregates the spec and implementation of the *Interprete Simple* and *Composite State* procedures, as well as the specs of the generic state machine structures (cmp. fig. 4.5).

We allow one to many SRSM processors, as at least the kernel shall be based on this pattern. Customization of processors for specific services is not desireable, but possible. Correspondingly, the cardinality of ISMC interpreters depends on the partitioning of native action subsets over the services, potentially (again, not desireable) resulting in as many interpreters as services, if every service component should require different native actions.

- *Generic Extension*
  Provides a set of *Basic Actions.* Conforming to the descriptions of 4.2.3.4, this action set realizes external procedures and functions assumed to serve on a level common to many applications of the domain, e.g. file operations, vector calculations for map matching, storage of configuration data entries, locating, timing and calculating amounts of money, but it may also realize access to essential cryptographic operations of some SAM, like de-/encryption and hashing/signature checking.

- *Application-specific Extension*
  Offers external actions realizing proprietary operations, e.g. message de-/encoding including object de-/serialization, operating database management, communications management (opening, using, closing channels), access to devices and sensor readouts.

  Both ISMC and SRSM components may refer to the included actions, both are *complemented* by the generic and application-specific extensions, resulting in a full, deployable RTE for the applications. While reuse is encouraged, if necessary, every service component may be supported by a dedicated extension.

- *Communications*
  Contains the *Message Queue* specifications and implementations employed by the kernel and components for interaction to assert uniform communications. Communications form a distinct conceptual package, because depending on the actual, underlying (hardware) RTE, the realization may need to comply closely to proprietary mechanisms, which we need to isolate from the generic elements.

  The message definitions distinguish *Infrastructure* and *Application Messages.* Infrastructure message specifications refer to the generic, underlying mechanisms of the HIRTE, i.e. interactions between kernel and components for management purposes – described in the context of the kernel modules in the next section. Messages introduced by the application realize interactions between the service components and potentially beyond to center systems. Handling of these message types by the kernel is supposed to be transparent on the service respectively application level.

- *Application-specific Business Logics*
  Composes the *Service* component specs and bodies. While, as stated in the previous chapters, the architecture avoids inheritance, services need to conform to a uniform component structure. This is achieved by the interfaces *ITrace* and *IRunControl*.

  An implementation of *ITrace* provides the method *LogState* (cmp. 4.2.2) to *feed* the current state configuration to the kernel, utilizing the static memory segment reserved for the respective component. Note that while the interface is defined for the service component, it is also used solely by that element. *IRunControl* answers execution instructions (see above) from the kernel.

  With these interfaces given, the realization of a HIRTE component may take different forms – according to the established patterns and beyond. *Component VCU* is a service implemented as VCU. It aggregates the application-specific structures as defined in 4.2.3: a *HIRTE Component* class and the *Task Attribute Set* record, with the *ISMC* record representing the state machine code defining the service's behavior. An interpreter is instanced from the *Generic HIRTE* package, input and output queues from *Communications* (see above). Similarly, the *Component SRSM* aggregates a *Task Attribute Set*, all *State*, *Transition Specifications* and associated structures as specified in 4.2.1, making up the behavior. Interpreter procedures for simple/composite states and the queues are taken from the generics and communications packages, respectively.

  Additionally, an *External SM* may implement any other – non-HIRTE, thus designated external – approach to component state machine patterns. As long as it conforms to or maps the defined interfaces, including the interactions via the queues and messages of the *Communications* package, the HIRTE application may integrate such a service.

5.1.4 Implementation Considerations

degrees of freedom

Chapter 4 lays a concrete foundation for the setup and the implementation of a HIRTE application architecture. Further context is established by the pattern of this section in fig. 5.1. Nevertheless, actual application modules still find degrees of freedom to emphasize varying elements of the HIRTE. When designing a solution based on HIRTE, important and closely related considerations will be the balance between

- state machine and action/operation logics, and

- native and external operations.

state sequences versus operations

The first point touches a fundamental question: which (sub-)processes respectively activities of the application are suited to adapt a state machine structure? It might become tempting – think of a pressing project schedule that comes to consider the modeling a hindrance for quick results (code) – to hide[2] significant parts of the logics in operations. During the course of this work, and explicitly with requirement II, we strongly advocated a state-based application design. In consequence, it is consistent to only exclude *transforming* (cmp. 2.1.1.1) parts with a clearly direct, functional and stateless relation between input and output from the automaton state set. Even in the case of a step-wise numerical computation with intermediate results, decomposition into a state sequence might be advisable to ensure reproducibility. The *Action* of 4.2.1.3 and *External Operation Set* of 4.2.3.4 established freedom from side-effects and variable access restricted to an attribute set, giving another distinction principle.

native versus external operations

An evident approach to the second point lies in the integration of device access APIs or drivers, OS functions and other third-party libraries. Usually, these kinds of modules have to be considered black boxes owing to legal reasons, with the external form as only option. Apart from that, it becomes a question of size and adaptability of the interpreter engine. Reducing native operations to an elemental minimum keeps the ISMC RTE lean, but requires foresight concerning the ISMC programs it is supposed to run, respectively which external operation sets they will supposedly need to access. In effect, this contrasts tailored, specialized application VCUs to full spectrum versions. The operational level of the corresponding service – does it realize a high-level business subprocess or a basic device management close to the hardware? – is a potential indicator for the adequate scale of an underlying VCU, as the degree of VCU specialization rises with a lower level.

Run versus Step

Finally, an implementation may have to assess the interface operations *Run* and *Step*. In the case of a VCU-based component, step semantics are explicitly defined. Finely grained for subtle control and scheduling options, they refer to a step of the interpreter and operations on its tape. If not handled, incongruencies might arise in heterogenous environments that combine different concepts of steps, e.g. VCU ISMC and SRSM, stepping through the machine's state set, or any other approaches focusing on transitions, actions, etc. We have to consider if step-wise execution is required for a given application or component, or if we may rely on OS threads and scheduling with plain calls to *Run*. In the former case, the high-granularity have to be mapped to the low-granularity step semantics to ensure homogenous execution and control respectively scheduling.

[2]E.g. from the perspective of a HIRTE State Tracer.

5.2 HIRTE Implementation Pattern

5.2.1 Abstract

The HIRTE's *implementation* view puts its elements in a realization context, i.e. a setup for instances of the framework pattern. With regard to the application, it describes the infrastructure roles and purposes immediately above the abstraction level of actual services. Consequently, it is made up of and – in the case of the kernel – refines the conceptual packages of the previous section.

5.2.2 Structure

Fig. 5.2 describes the structure of the implementation pattern.

5.2.3 Collaboration

- *HIRTE Kernel*
  To fulfill its supervisory purpose as described in 5.1.3, the kernel module composes a set of subcomponents.

  Component Manager

  A *Component Manager* serves as registry and configurator of the services, prominent features depending on the type of implementing components. Generally, the manager administrates service priorities, message or event subscriptions compliant to a publish-subscribe mechanism (cmp. e.g. distribution patterns in [Do03]) and other properties related to the service behavior and interactions. In the case of VCU modules, service ISMC programs may actually be replaced or modified during run-time[3], resulting in necessary (re-)registration of the potentially updated service characteristics.

  Scheduler

  The primary assignment of the *Scheduler* lies in the practical exercise of the automaton's step semantics in the single-thread scenario (see 5.2.4 below). Observing the above configured priorities, it invokes the associated VCUs' ISMC or SRSM interpreter *Step* methods. If an underlying OS provides threads of execution and thusly a proprietary scheduler, the HIRTE scheduler module reverts to managing the corresponding interface as a secondary option, e.g. starting, suspending and stopping the service components.

  Communications Broker

  The *Communications Broker* represents a controller for the message-based interactions between the HIRTE elements. For this purpose, it handles all *Message Queues*. This implies that communications between

[3]At first glance, this may seem to violate our staticality constraints. However, changes to the overall configuration of the application must only be conducted during a maintenance superstate – cmp. *Service and Diagnosis* in chapter 2.3.8.3. Thus, during regular machine operations, priorities may remain fixed.

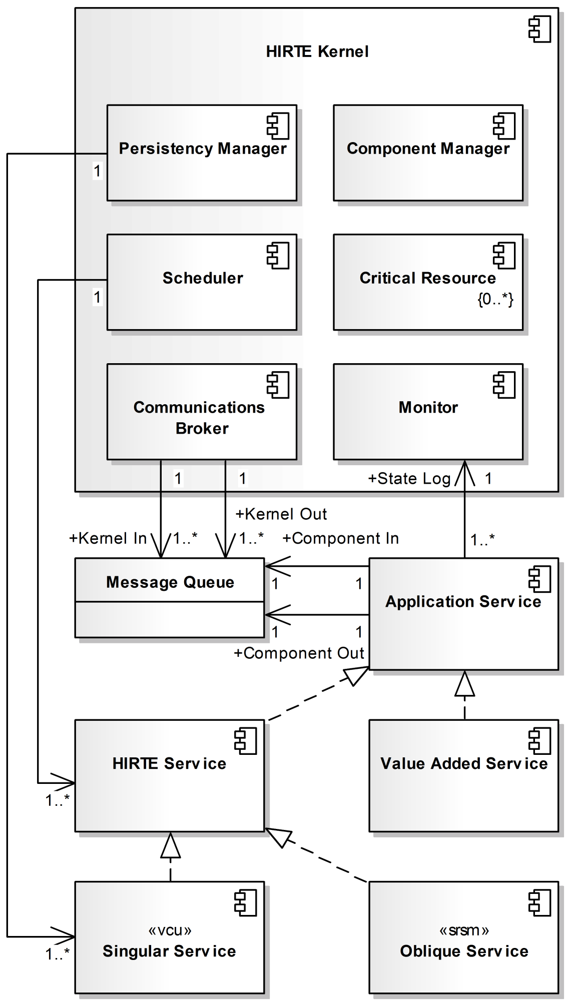


Figure 5.2: HIRTE Implementation Pattern Structure

a component and the kernel (via *Infrastructure Messages*, cmp. previous section) and broadcasts, as well as information exchange between service components (via *Application Messages*) use the broker as mediator, or switchboard. A component addresses another by complementing a message with the receiver's ID and sending it to the broker, which receives it in one of its *Kernel In* queues. Assigning it to the *Kernel Out* queue with the corresponding ID, the broker forwards the message. Again, this aims at facilitating controllability, in this case by enabling the kernel to regulate message distribution and frequency in the interval between reception and dispatching.

Monitor

Aggregating all information about the system's state configuration, the

Monitor realizes a passive module. For the service components, it assumes the role of current *State Log*; for the kernel, it incorporates a *State Tracer* in conformance to the elements of the State Tracer pattern of 4.2.2. Additionally, it may observe any other mechanism or resource deemed relevant for the stability of the system's activities. The kernel may then further collate, analyze or synthesize the collected information to assess conditions and tendencies (cmp. 3.1.2), and take proper measures, e.g. signaling center components.

Persistency Manager

For VCU components, the *Persistency Manager* enforces requirement VII (persistency of activities): it copies the ISMC tapes – representing the complete state of a VCU program (cmp. 4.2.3.2) – and message queue contents to a static storage. Triggering of this action is synchronized with the scheduler, which may employ the step semantics and processing to define such a transaction. Its granularity has to align to the service activity and will potentially range from corresponding to the interpreter steps, in the case of crucial components demanding high integrity of processes, to service states or even just specific events, e.g. start/end of interactions or computations, reported to the monitor. If all tapes and queues were successfully saved, i.e. stored integrity-checked, the transaction is commited, otherwise the system may rollback to the previous state configuration to maintain processing consistency.

Critical Resources

Especially in deployments without an OS, the kernel itself may be compelled to manage a set of *Critical Resources* – storage segments, communications interfaces, or generally limited capacity devices and elements providing some kind of write operation for the service components. Access interaction is handled by the communications broker and infrastructure messages (see above), management and mutual exclusion by the corresponding kernel resource component.

HIRTE Kernel versus Hypervisor

Put in the already discussed context of virtualization (see 4.3), the approach to control a set of virtual machines, in the given case the VCUs, by a superordinate instance like the HIRTE kernel, resembles the *Hypervisor* concept (cmp. e.g. [Go73] for the related *Virtual Machine Monitor*). However, our differentiation still stands: the common solutions enable complex, universal multi-OS configurations, whereas the HIRTE kernel specializes on the high-integrity telematics domain with a dedicated approach regarding design, implementation and especially constraints. Furthermore, it exerts direct, active control of the VCU machines and explicitly provides lines of communication between the components, which usually depend on each other to realize an overall application. Present hypervisors rather focus on hardware resource sharing under strict partitioning of OSs and applications.

- *Application Service*

Designates a general service component in the context of a HIRTE architecture. As the most basic of service abstractions, it features a *Component Out* message queue to send messages to or via the kernel and a *Component In* queue to receive messages from the kernel, and from other components via the kernel. For the implementation of the *ITrace* interface (see framework pattern of 5.1), it is associated with a *State Log* memory segment of the *Monitor*. Thus, an application service is expected to register its current state with the kernel.

- *Message Queue*
  Represents a structured container to manage infrastructure and application-specific messages exchanged by HIRTE components and kernel, as described in 4.2.1.3 and 5.1.3. With the kernel as central hub, a pair of dedicated queues is assigned to every service component for inbound and outgoing communications.

  To effect rapid persistency, a queue is supposed to store messages in a serialized representation format, i.e. as (binary) strings, regardless if the message is conveyed to a remote RTE or received and processed locally. As attaining a persistent system configuration may be time-critical (cmp. e.g. 3.1.4), this anticipates serialization by the *Persistency Manager*, relieving it from having to iterate all queues for this purpose.

- *HIRTE Service*
  A refinement of the *Application Service*, in addition to conforming to a state machine structure respectively behavior, it implements the *IRunControl* interface (cmp. 5.1.3). In consequence, the HIRTE service constitutes a subject for the kernel's *Scheduler*, submitting to a higher, or active, degree of controllability than the general service component.

- *Singular Service*
  Realizes a *HIRTE Service* as VCU with high-integrity requirements, demanding transaction-based persistency of its state configuration. To this end, the *Persistency Manager* may directly access the ISMC tape.

  This kind of service realizes a vital element of the application, with an activity and state configuration that cannot be reconstructed from other component states and is not implied by other modules, e.g. sensors or modems. As an example we find a complex real-time rating and payment process with intermediary, transient results.

- *Oblique Service*
  In order to propose some distinction between scenarios for the adaption of either a SRSM or ISMC VCU, we suggest to employ the exhaustively controllable VCU for *Singular Services* (above) and SRSM-based components for dependent, oblique services. These may not necessarily be

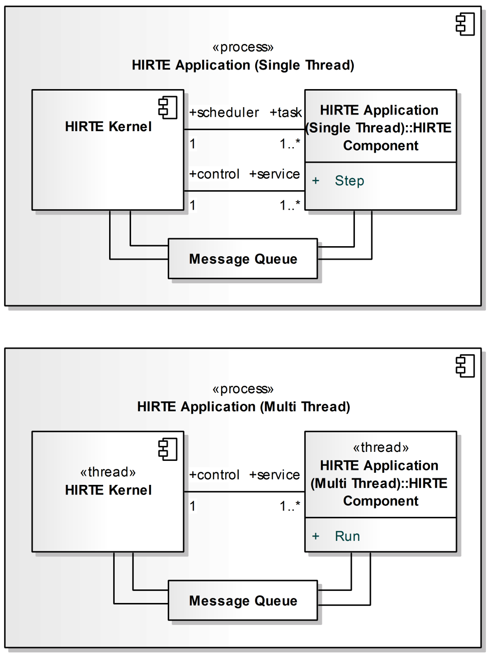

«process»
HIRTE Application (Single Thread)
HIRTE Kernel
+scheduler
+task
1
1..*
+control
+service
1
1..*
HIRTE Application (Single Thread)::HIRTE Component
+ Step
Message Queue
«process»
HIRTE Application (Multi Thread)
«thread»
HIRTE Kernel
+control
+service
1
1..*
«thread»
HIRTE Application (Multi Thread)::HIRTE Component
+ Run
Message Queue

Figure 5.3: Uniform Handling of Single- and Multi-Thread HIRTE Configurations

less crucial to the application, but their state can be derived from other components, and is therefore not submitted to the *Persistency Manager*. However, they need to enable scheduling by the kernel.

Typical examples would be services that manage devices like a GNSS sensor – polled in a frequency set by the kernel – or CN modem. The relevant information is not held by the service component. Instead, e.g. after a shutdown, it is merely reread[4] from the originator.

- *Value Added Service*
  Without the need for active intervention (beyond plain termination), these services do not contribute to a crucial application, merely running in its RTE (cmp. 3.1.3.1). Consequently, the kernel requires their state log to notice and analyze potential disruptive incidents, e.g. resource access collisions.

5.2.4 Single-Thread and Multi-Thread Environment Variants

coping with rudimentary RTEs

For commercial acceptance in the context of the ETC use case, the question

[4]Assuming proper handling that only deletes data on acknowledgment of processing by the client.

for capabilities of an RTE platform may become crucial: tenders might require low-cost hardware, dramatically restricting options of an OS with sophisticated process/thread management. While this may come down to a single-process single-thread RTE without scheduling, the HIRTE still has to cope with the concurrency demands of the ETC business process as established in 2.3.8.

flexible deployments of VCUs and SRSMs

The HIRTE architecture is expressly intended for both complex OS as well as minimum RTE environments (e.g. smartphones in contrast to ECUs). Thus, it provides facilities for both scenarios. As a VCU realizes a proprietary RTE, it is the obvious choice for deployments with only a rudimentary OS respectively RTE. However, like shown in the previous section, it is not limited to this assignment (also see next paragraph). The interpreter faculties of the SRSM component refer to structures declared in its source code, without the explicit program/interpreter separation of VCU ISMC. One consequence of these approaches is the differing granularity of step semantics, relevant for program monitoring, but also for scheduling of component program execution.

thread handling

A distinctive feature of complex versus plain RTE is thread handling. For the domain considered in this work, the *POSIX* standard (cmp. [IO08]) offers the currently prevalent view on thread-based concurrency. Pragmatic considerations regarding HIRTE, however, have to accept the fact that industral practice reveals inconsistencies and ambiguity in the "POSIX-compliance" of available RT-OSs (examples in 2.1.2.3). One proper answer lies in the option of forgoing the OS mechanisms altogether, relying on a set of VCUs instead. This decision is important to illustrate that the selection and adaption of the corresponding HIRTE elements is not necessarily dependent on OS features only.

dual setup

Regardless of the design decisions, HIRTE offers two configurations for single- and multi-thread scenarios, illustrated in fig. 5.3. Both assume the HIRTE elements to run in the context of an application, itself a *process*, i.e. an execution unit with some (not necessarily exclusive) address space.

single-thread specifics

The single-thread configuration specifies the *HIRTE Kernel* as *scheduler* and *control* of the associated *HIRTE Components* realizing the application's services. Components become instances of e.g. the class as defined by 4.2.3, implying the ISMC interpreter, external operation and task attribute sets. Input and output message queues connect all components with each other via and with the kernel. Sensibly, the kernel will implement some primary, *main* operation of the application process. Corresponding to its *scheduler* role (and module), the kernel regards the components as *tasks*, calling their *Step* methods in an execution loop, conforming to a scheduling scheme (cmp.4.2.3.4, 4.3). Beyond that, between scheduling cycles, the kernel manages and controls the *services* as described in the section above, employing its remaining modules for monitoring, messaging, persistency etc.

multi-thread specifics

A multi-thread configuration sees kernel and components as *threads of execution* running in the context of the application process. Each thread implements some *Run* method, with the kernel responsibility now reduced to managing its modules. Accordingly, the *scheduler/task* association does not apply, the duty now falling to the OS. In the implementation, e.g. the *HIRTE Component* class of 4.2.1 will extend some proprietary OS API class, or implement some interface, structure or functions. A risk assessment for an actual deployment in question will have to evaluate if this delegation is acceptable for the HIRTE application supplier.

dual validity

The high degree of structural congruency in the model of fig. 5.3 emphasizes the flexibility of the approach. Most of the elements introduced in chapter 4 are valid and serviceable in both configurations.

5.3 HIRTE Application Pattern

5.3.1 Abstract

A view on the *application* of HIRTE produces a pattern that assembles a set of services making up an actual ETC OBE software. Concerning the application, there is not *the one* pattern – we sketch an original solution based on the insights and suggestions of this work, loosely oriented toward the requirements of EETS (cmp. [EU09]). For other system deployments, this structure may differ. In the process, we apply the setup of approach III; the interoperability scenario of 3.1.3.3.

As hardware execution environment or OBE, here we consider a smartphone. Irrespective of the actual type and OS, this platform touches many of the topics discussed and is thus a rewarding subject. It is a representative telematics platform, integrating CN services and the capability to run complex software, with various additional interfaces for local machine-to-machine interaction (e.g. Bluetooth, WLAN) and to other devices (e.g. memory cards, automotive networks and sensors). As a generally non-dedicated environment, unlike a proprietary ETC OBU, other applications are likely to run concurrently to our ETC software, stressing the reliability aspect of a safe application in an unsafe RTE and also corresponding to our approach I to economy with an open infrastructure (cmp. 3.1.3.1).

5.3.2 Structure

Fig. 5.4 presents the structure of an application pattern.

5.3.3 Collaboration

- *Universal ETS App*
  Composes a set of services and a management component for the realization of an ETS business process. As such, it stages an ETC OBE software to begin with, constituting a universal foundation for the application. The degree of this generality – and thus the variety of actual,

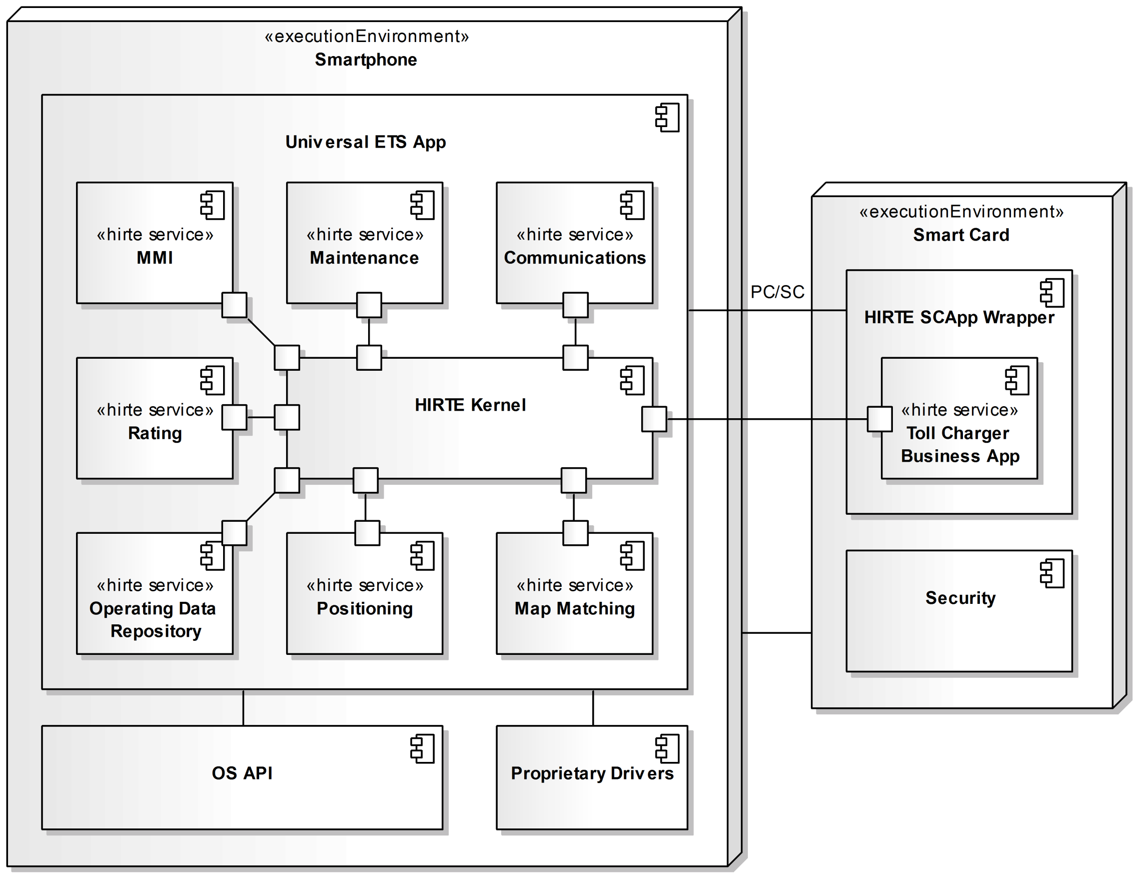

«executionEnvironment»
Smartphone
Universal ETS App
«hirte service» MMI
«hirte service» Maintenance
«hirte service» Communications
«hirte service» Rating
HIRTE Kernel
«hirte service» Operating Data Repository
«hirte service» Positioning
«hirte service» Map Matching
OS API
Proprietary Drivers
PC/SC
«executionEnvironment»
Smart Card
HIRTE SCApp Wrapper
«hirte service» Toll Charger Business App
Security

Figure 5.4: HIRTE Application Pattern Structure

realizeable applications based on it – depends on the qualities and quantities of composed service components.

To integrate the component implementing the operator-specific business process logics of the ETS (see below), the universal app connects to a smart card via e.g. PC/SC (cmp. [PC05]) or similar interface.

- *HIRTE Kernel*
  After declaring its conceptual role in the set of HIRTE elements as controller and manager of the generic RTE, its composed elements and duties to other components of the implementation, in the application the kernel serves as the unifying component of all services. Thus, here it assumes the role of the *OBE Management Software* of 3.1.3.3.

  The kernel connects all services via the message queues and associations for scheduling and persistency as specified in the previous section 5.2. In fig. 5.4, this is depicted implicitly by the associations and ports that may be used to aggregate the messages employed to offer and request the services.

- *MMI*
  Displays states and information relevant to a user, e.g. rating results, overall operativeness, and handles user interaction. In a smartphone environment, this component very likely relies heavily on proprietary GUI libraries of the OS, integrated in the context of actions (SRSM) or external operations (ISMC).

- *Maintenance*
  Is activated when the application enters the states *Installation and Commissioning*, *Service and Diagnosis* or *Decommissioning and Deinstallation* of the ETC OBE software business process (cmp. 2.3.8). Stopping or locking all other, unrelated services and interfaces, maintenance interacts over a dedicated protocol in close coordination with the *Communications* service. In the case of severe failures and a requisite visit to a repair shop, the protocol will rely on short-range media like Bluetooth or even cable; for regular maintenance, on CN OTA solutions.

  The maintenance service allows access to and updates of crucial elements of the software, i.e. the executable itself, ISMC programs and smart card applications. Additionally, it is used for recovery of system logs after critical failures prevent regular remote log analysis.

- *Communications*
  Provides access to the interfaces of long- and short-range communication devices, and manages them, where applicable. The degree of control the software may exercise on these resources depends on their exclusiveness: it has to compete for smartphone internal services like GSM- or UMTS-based data transmission, WLAN and Bluetooth, but may command any additionally connected units discretionary.

  Management of communications implies message en- and decoding, corresponding to the proper transmission formats required by the interfaces, and connection (re-)establishment and termination (session management). For incoming messages or connection sessions, other services may register themselves as receivers via the kernel.

- *Rating*
  Applies the tariff scheme (cmp. 2.2.1) provided by the *Operating Data Repository* to a toll atom identified and passed by the *Map Matching* service, factoring in the current vehicle configuration and local time, if required. In the process, the rating service generates a billable entity, similar to a CDR. This, in turn, is handed to and processed by the smart card application (see below).

- *Operating Data Repository*
  Realizes the *active data storage* aspect of 2.3, i.e. it stores and also actively manages map respectively geometrical toll atom data, the tariff scheme, configuration and set of receipts. This implies checking the stored datasets for validity periods and requesting *Communications* to contact the center services for updates on a regular basis. Locally, the repository ensures that crucial data is filed in static storage memory spaces.

  For other services, the component provides reporting and location-based, spatial querying functionality, accessing map segments for *Map Matching*, tariff scheme entries for *Rating* and storing configuration data for *Maintenance*. Implementation of the low-level storage operations may utilize the OS's file system (actions and external operations, see above), but will also integrate other memory devices for safe respectively static storage.

- *Positioning*
  Determines the vehicle's respectively smartphone unit's position using a sensor fusion of the integrated GPS receiver, GSM signals and gyroscope, if present. The ETC system may demand additional, external sensors like tachometers that require connection of the smartphone to a vehicle subsystem (e.g. via the gateway of 2.1.1.3). For positional data, other services register with their required frequency of updates via the kernel.

  A risk analysis has to determine whether the service can rely on OS-native interfaces for positioning, or needs to implement its own, proprietary sensor fusion algorithm for reproducibility and verification. In the case of OS modules, the service at least has to continuously gain a measure of their quality of service and indicate potential problems.

- *Map Matching*
  Maps the output of the *Positioning* service to geometries referencing or representing toll atoms provided by the *Operating Data Repository.* The complexity of the atom geometries as well as the matching algorithm implemented by this service again depends on the degree of reuse of facilities provided by the smartphone native libraries and services. If their quality is considered sufficient (risk analysis), the ETC operating data toll atom set may be reduced to a layer over e.g. a navigation service – the application's service component measuring its reliability.

- *HIRTE SCApp Wrapper*
  Realizes the specification of a HIRTE component (cmp. 4.2.1 and 4.2.3) as smart card platform-specific application. The message queue interfaces are mapped to APDUs respectively PC/SC.

- *Toll Charger Business App*
  The consideration of EETS in the given pattern implies adapting the distinction between *toll charger* and *EETS provider*. With this service, the EETS provider, responsible for the technical implementation and operation of the system, supports the business processes stipulated by the toll charger, in turn responsible for the economical, legal and organizational aspects.

  Consequently, this component integrates all other services in its implementation of the business process (for detailed steps of the process refer to 2.3.8). Corresponding to the concept of 3.1.3.3, it is kept lean, limited to the proprietary arrangement and conducting of the activities by message interaction via the kernel. This implies handling charging/payment (cmp. 2.3.2) based on *Rating* output, and enforcement responses (2.3.5), as the associated structures and records, e.g. accounts, are likely to be provider-specific and also sensitive concerning security.

- *Security*
  Provides functions for en- and decryption, hashing, signing and authentification as described in 2.3.6, encapsuled as SAM.

- *OS API*
  Provides a variety of interfaces and modules – depending on the actual platform – from low-level device drivers to application frameworks. Whether an OS API element may substitute (parts of) a HIRTE service has to be subject to a risk analysis and evaluation (for the analysis template see 3.1.1.10).

- *Proprietary Drivers*
  Device drivers for additional, external sensors, automotive networks, media etc. not provided by the smartphone platform.

5.3.4 Setup and Deployment Alternatives

adaptability

Selected as representative application type of this work (cmp. 2.2.3), the pattern of fig. 5.4 sketches a thick client solution on an open platform. We can illustrate its adaptability to other setups and deployments with a short excursion to a thin client application and a dedicated OBE platform.

thin client HIRTE

Pursuing the thin client approach reduces the demands to the RTE platform, but shifts them to the center and especially to the communication infrastructure: the OBE merely collects positions and passes them to the center systems for further processing. For the application pattern the client complexity reduction means

- optimizing *Communications* for packed position track encoding and transmission,

- removing the *Rating* service,
- removing toll atom data structures from the *Operating Data Repository*, enabling it to temporarily and securily store position tracks (if CN transmission fails, *Maintenance* has to recover the full tracks),
- either removing the *Map Matcher*, or reducing it to border detection of toll charger respectively operator areas and
- removing states processing an enforcement transaction, charging/payment and associated local accounts from the *Toll Charger Business App*, replacing these activities with position handling (personalization, encryption of tracks or tours).

In summary, HIRTE applies to a thin client software without reservation: the reliability and economy requirements are still valid, reproducibility of activities in case of problems is still crucial.

HIRTE on dedicated OBE

Different approaches to ETC OBE platforms were described e.g. in 3.1.3.2 and 3.1.3.3. Similar to the thin client architecture reducing software and distributed process complexity, a dedicated platform requires less effort to operatively handle than the open variant. The pattern's setup projects on a single application OBE without significant alterations. Resources are exclusively available for an ETS application; concurrent modules and competition for access do not apply. However, internal complexity of the service components may rise, as the dedicated platform is likely (based on our industrial experience) to offer a plain OS only, or none at all. In consequence, actions and external operations of HIRTE components may not rely on sophisticated application frameworks. Still, the consequent HIRTE-proprietary implementations benefit controllability.

5.4 Chapter Conclusion

abstracting to the HIRTE ...

The elaboration of this work lead us from a treatise on the high-integrity telematics domain of ETC over a thorough analysis of its aspects and processes – focusing on economy and reliability – to the HIRTE building blocks of chapter 4. The ETC use case served as a representative archetype, featuring properties like vehicle-embedded distributed deployment, cash value equivalent OTA transactions and complex liable activities that need exhaustive monitoring. From this concrete example, we derived a set of requirements that allowed us the directed, systematic construction of the HIRTE.

... and coming down to an application

With the HIRTE core elements, we purposefully reached an abstraction from the ETC domain. This chapter now gradually leads the focus back to the application level of ETC in three steps:

1. The *framework* pattern groups the original HIRTE building blocks, assigns roles and relations between them. This way, it provides a blueprint after which the reference implementations may be organized in libraries of an application framework. Finally, the pattern assists reasoning about the design of an actual solution concerning the structuring of its states and operations as defined by the HIRTE.

2. The *implementation* pattern details the kernel and the internal services it implements to manage a HIRTE software. Furthermore, a realization hierarchy of the service components is given – depending on the context and setup of an application, a HIRTE implementation allows different levels of module integrity; ranging from merely monitored to tightly controlled integration.

3. The *application* pattern then specifies the structure and services of a thick client ETS solution on an open platform. In the process, it picks up on an original concept of smart card integration. As declared in the previous two patterns, the HIRTE kernel now represents the integrating and controlling instance of all services. These in turn are specified as HIRTE services, i.e. they need to conform to the structures and interfaces of the implementation pattern. Selection of a variant – SRSM or VCU, single- or multi-thread – is left to an actual realization, which, adhering to the ETC OBE business process (2.3.8), will then reflect the distinctive ETC aspects of the domain introduction (2.3).

coming full circle

By setting the generic HIRTE elements in the context of an ETC application, this work is finalized by coming full circle to an answer to the initial question of a reliable and economic telematics software solution in the domain of ETC. The provided three patterns illustrate the consistency between conceptual elements and application, allowing the pursuit of congruency of ETC aspects (chapter 2), the genuine requirements (chapter 3) and HIRTE fundamental patterns (chapter 4) with an application solution design.

Six

Conclusion and Outlook

... in which we discuss and summarize the results of this work, give some comments and suggestions for their application, and encourage potential future research and development.

6.1 Approach Evaluation and Achievement of Objectives

For a discussion of the results of this work, [HMPR04] provides orientation. Focusing on design science, it reasons about a framework for information systems research. We can apply its guidelines to gain a measurement and understanding of validity and value of our approach and obtained artifacts.

6.1.1 Concerning Guideline 1: Design as an Artifact

"Design-science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation."

primary results and their structure

We regard the building block patterns of chapter 4, together with their integrating patterns of chapter 5, as the primary results of this work. For the fundamental patterns, we specify a *model* defining the reproducible structure of each suggested solution part it represents, describe the *collaboration* of its elements and give a manifest *reference implementation* in Ada. The application architecture patterns specify the purposeful composition of the original elements, in the process assigning roles to the corresponding components to realize an actual solution for the mission-critical telematics respectively ETC domain.

- The artifact *fundamental pattern* is a blueprint for application components in the given or a related domain. Adapting to the SRSM pattern of 4.2.1, a component may conform to a hierarchical state machine structure and behavior, maintaining a static memory footprint. The pattern and interpreter automaton specifications of the VCU component of 4.2.3 enable the design of a high-integrity network of lightweight virtual machines controlled by programs representing encoded state machines. With the introduction of DSMFs in 4.2.4, a VCU program may selectively delegate fragments of its processing to other RTEs. Generally, a focus on state automaton formalisms facilitates systematic monitoring and reproducibility of all activities – supported by a state tracer module (4.2.2) that allows the instrumentation of a system for the purpose of state configuration tracing.

- Preparatory to the DSMF, the formal introduction of the *self-contained state* in 4.1.3 augments and extends the concept of state automatons toward distributed execution.

- The complementing artifact *reference implementation* removes ambiguity from the presented models and pragmatically transports the intended characteristics of an instantiation, while immediately warranting applicability. Sources as given in listings 4.1 to 4.16 may directly implement the core of a safe telematics application framework or serve as point of origin for integration, refactoring or translation to other languages.

- The integrating artifacts of the *application architecture patterns* describe the views on a valid HIRTE system architecture. By separating the concerns of *framework*, *implementation* and *application*, we open the overall system setup to reuse and adaption. The framework pattern arranges the above basic pattern components into generic and application-specific as well as infrastructure and application elements. In the implementation pattern, roles and scope of infrastructure and application components are refined, with the application pattern mapping actual ETC services to the elements. Each of the patterns allows substitution, specialization and refinement of the composed components inside the conceptual boundary of HIRTE, as given by the requirements of 3.1.2, 3.1.4 and constraints of 3.2 and 3.3.

secondary results

With regard to methods, we present a number of secondary results; byproducts of the elaborations on HIRTE. The *hazard analysis of an ETC system* (cmp. 3.1.1.1 to 3.1.1.9), leading from FTA to the *generalized classification scheme for mission-critical problems* in 3.1.1.10 is valid for future deployments in the telematics domain. Likewise, the derived requirements I to VII provide a *guideline for the specification and development of high-integrity telematics solutions*. As a practical solution complementary to the HIRTE approach, 3.1.3.3 describes a *smart card integrated interoperability concept*.

6.1.2 Concerning Guideline 2: Problem Relevance

"The objective of design-science research is to develop technology-based solutions to important and relevant business problems."

the domain of ETC ...

Chapter 2 introduces the domain of this work. In the case of ETC, it is sensible to actually establish the domain from a software system view, as available works on the topic either focus on engineering/hardware, lack the necessary substantial operational experience or simply do not exist. Furthermore, the differentiation from related domains – like the discussed parents of automotive and telematics systems – yields sufficiently significant unique characteristics of ETC deployments (cmp. 2.3, 2.4).

... and what it stands for

At the same time, ETC software architectures generally represent a kind of large-scale, mission-critical telematics system that is currently just emerging – with still only one major GNSS/CN ETC installation, the ongoing discussion of the German *Elektronische Gesundheitskarte* in eHealth, promotion of Smart-Grids in renewable energy systems. The specific or even unique requirements regarding economy and reliability justify a dedicated approach. Summarized, we find an original combination of

- a highly complex system of distributed and center software components interacting over telecommunications interfaces,
- managing, handling and measuring cash-equivalent services reaching a turnover with the magnitude of billions of Euro per year and
- penalty-relevant service level agreements with third parties[1] besides operator and customer: government authorities, infrastructure and resource providers.

domain-specific questions and needed answers

Problems, for whose solutions especially the software can contribute, arise from the combination of *heterogenous platforms of the distributed components* and *liability of an operator for steady charging and accurate accounting.* Due to OBE elements eluding application control (cmp. 3.1.2) – in the future even more pressing, when migrating from dedicated devices to open platforms like smartphones –, the answer shifts from the common solutions of e.g. redundancy to a *fault-aware, safe application in an unsafe environment.* From the view of an OBE software supplier with the significant difference of abandoning reliability in the sense of global high-availability in favor of enforcing local, component-wide controllability and reproducibility of activities. The operator has to be enabled to rapidly identify problem causes and actively prove the correctness of its very own component, including interactions with the system environment, to keep it clear of claims of compensation in the case of aggregated quality of service transgressions.

[1]An important distinction to the admittedly sophisticated but actually rather noncommital common services of telco operators.

Recalling the generic telematics infrastructure of fig. 2.6, the software components treated by this work are fittingly situated in the deployment to provide answers to the raised problems.

6.1.3 Concerning Guideline 3: Design Evaluation

"The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods."

custom-tailored architecture

The primary artifacts of this work, the HIRTE patterns 4.2.1 to 4.2.4 and 5.1 to 5.3 with their reference implementations, were specifically constructed and tailored to the reliability and economy requirements derived from the domain. Additionally, they adhere to a set of constraints regarding model and programming (cmp. above 6.1.1).

predetermined evaluation

In consequence, software architecture evaluation methods like ATAM (cmp. [CKK02]) would lead us to cyclic dependencies of inferences: its utility trees should resemble refinements of the stated requirements and constraints, the scenarios closely relate to or imply the aspects of 2.3. The evaluation conclusion would be predetermined.

evaluation by implementation

As an alternative, the artifact reference implementation assumes two additional roles:

- *validator* – as described in 4.3, both the structure of application components conforming to the HIRTE framework and the HIRTE infrastructure itself (SRSM elements, ISMC interpreter) adhere to a systematic state automaton configuration. This facilitates a directed construction of test components a) covering all features of the state machines supported by the SRSM and ISMC patterns, and b) covering all states of the ISMC interpreter. By analyzing the resulting traces (test application and infrastructure elements) and comparing them with the model artifacts, we verified that the realization behaves as intended, as well as that the model describes the intended system.

- *proof of concept* – on a more general, for all intends and purposes conceptual level, representative application modules may employ the framework for realizations of domain-specific use cases. In effect, such implementations prove the adequacy of the offered solutions quasi-empirically. Similar to the evaluation of the architecture, an empirical evaluation of the efficiency of the implementation's structures should be considered redundant[2]: the constructive approach (see below) ensured compliance with the domain constraints of language subsetting, static memory footprints and deterministic processing in a reproducible way.

[2]Referring to the generic level; evaluations of potential deployments and specific underlying RTE platforms notwithstanding.

6.1.4 Concerning Guideline 4: Research Contributions

"Effective design-science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies."

establishing the domain itself

The initial contribution of this work is the elaboration and introduction of the mission-critical ETC telematics domain – in this case, the area itself – into the discipline of software engineering. This includes a set of results that may be directly applied to system architectures of industrial projects and scientific works, most notably the *aspects* of 2.3. Analysis, classification and structuring of planned or existing systems based on this scheme of cross-cutting concerns will yield evidence of its completeness and significance. Likewise, the *hazard classification scheme* of 3.1.1.10 directly allows alignment and comparison with an actual deployment and its components.

micro and macro level solutions

The software architectural contributions present views on two different levels of abstraction, corresponding to the requirements stated for the results of this work: globally, economy and reliability, refined into seven concrete properties to achieve these goals. Consequently, on a *micro* level, we provide solutions for safe, high-integrity components in a distributed deployment that conform to a state automaton configuration. The *macro* level synthesizes these elements into an application architecture, arranging them in specific roles. All levels and results focus on the previously defined domain; where necessary, they were differentiated from or compared to potentially related works, justifying and establishing the unique approaches of the HIRTE.

applicable artifacts

Each of the fundamental HIRTE patterns of *SRSM*, *State Tracer*, *VCU* and *DSMF* were designed to be serviceable and valid in as well as beyond the application architecture pattern context of chapter 5. Hence, complemented by their reference implementations and in the sense of the above proof of concept principle, they stand immediately available for empirical or other methods of verification. The same holds true for the macro level patterns, with the additional, initial effort of composing a full application.

6.1.5 Concerning Guideline 5: Research Rigor

"Design-science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact."

approach per task

Over the elaborations of this work and the successive development of the HIRTE, we applied the following methods to approach the different questions and emerging tasks.

- *Comparative* – chapter 2 contrasted related domains and the mission-critical telematics discovered in the ETC use cases. This comparison allowed a differentiation and the description of the domain's original aspects, structure and processes.

- *Analytical* – chapter 3.1.1 applied a FTA to the previously introduced domain and its elements, allowing purposeful deconstruction that resulted in a framework for risk-classification and requirements.
- *Constructive* – chapter 3.1.3 systematically developed approaches to infrastructure usage optimization, producing concepts of solutions in the telematics domain that aim at cost-efficiency.
- *Inferential* – from the results of the risk and economy considerations, we derived a set of seven requirements for a safe and economic software architecture (3.1.2, 3.1.4).
- *Comparative* – based on our specific requirements, we considered common, potentially related concepts like redundancy and fault-tolerance, as well as solutions like JavaVMs and established patterns for state automaton implementation. All options were reasonably discarded as unfit for our specific domain (throughout chapter 3).
- *Comparative* – unlike other works, we consider the implementation a crucial and inseparable element of a software architecture proposal (at least in the mission-critical telematics domain). Consequently, 3.2 discussed options of languages for our domain and selected a subset of Ada.
- *Constructive* – conforming to the given domain in general and specifically to the introduced requirements, chapter 4 described architectural building blocks for mission-critical, distributed components. To provide an exhaustive specification, they are composed of a model, collaboration and Ada sources, which adhere to the constraints of 3.2 and 3.3.
- *Empirical* – the reference implementations were used to test and validate the results of chapter 4.
- *Synthetical* – chapter 5 assembled the fundamental building blocks to an application architecture consisting of framework, implementation and application views.

6.1.6 Concerning Guideline 6: Design as a Search Process

"The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment."

steps of the elaboration

The primary results of this work as noted above are the result of a thorough development,

1. starting with the exploration and definition of the problem domain, its representative properties and constraints (*ETC aspects* and *business logics*),

2. continuing with an examination and elaboration of the characteristic details directly related to the initial questions of reliability and economy (*FTA*, *infrastructure usage optimization*),
3. deriving the desired properties of the result artifacts (the *seven requirements* and *complementing constraints*),
4. selecting the appropriate methods and tools for their construction (*state automata*, *UML model patterns*, *Ada*) with respect to specific limitations (*static implementability*, *language subsets*),
5. extending the methods where necessary (*self-contained state*, *stereotypes*) and
6. finding a suitable pattern set that – besides satisfying the domain's demands – allows universal application inside of the boundaries of the given domain.

6.1.7 Concerning Guideline 7: Communication of Research

"Design-science research must be presented effectively both to technology-oriented as well as management-oriented audiences."

science and industry

In the given case, this dissertation thesis was chosen as vehicle for formal presentation and introduction of the results into the scientific community. At the same time, the HIRTE will serve as core of a telematics framework product at omp computer gmbh in Paderborn, Germany.

technology and business

The initial, general requirements of reliability and economy reflect different views on the HIRTE architecture: the creation of reliable systems assumes a technology-oriented perspective, while – at least at first glance – economy should concern the business level of a product respectively solution development. In this work's chapter 3.1.3 however, we presented immediate technical solutions to an economical question. From a more abstract point of view, especially our concept of exhaustive (white box) reproducibility of processes, activities and interactions for the purpose of rapid troubleshooting becomes as valuable as the penalties warranted for system downtimes may be costly.

Thus, the HIRTE and its components can have a significant impact on both technical and business levels of a system.

6.2 HIRTE as a Framework

concrete artifacts

The HIRTE patterns in union with their reference implementations (cmp. chapters 4, 5) present concrete artifacts for the design and implementation of an OBE software. Component models of an application may be fashioned after the SRSM or VCU schemes, their Ada source integrated and refined into service realizations.

intuitive attempt

Use of the term *framework* is commonly rather intuitive and sometimes arbitrary. With the application architecture patterns, the employment of HIRTE in the building of a software system was described in detail: a reusable abstraction sets the scope, setup and collaboration for a realization. However, an explicit consideration of the framework character of the content of this work seems indicated.

framework characteristics

From [Ri00] we can derive some manifest characteristics to check the HIRTE elements for:

1. HIRTE kernel and component collaboration implies an *inversion of control* from application program to infrastructure – the HIRTE determines basic interactions and the underlying program flow.

2. The HIRTE infrastructure with the state automatons and kernel services defines a *default behavior*.

3. To implement an actual application, the HIRTE structures are *extensible*. Note that extensibility mechanisms based on the safe reference implementations do not support common object-oriented concepts like overriding or inheritance. Instead, it specifies the necessary interfaces and the run-time environment for applications and their services.

4. The crucial modules of the HIRTE – generic SRSM structures and VCU machines – are *not* meant for liberal *modification*. Instead, they are explicitly partitioned into core respectively generic and application-specific elements (cmp. e.g. the framework pattern of 5.1).

automotive similarities

Likewise, recalling 2.1.1.4, we can substitute the subjects of the identified basic aims of automotive frameworks with our VCU approach: HIRTE provides a RTE respectively middleware, unifies the communications between VCUs with the corresponding kernel management component, defines an interface and implementation rules for VCU applications, facilitating portability of the ISMC programs over systems that implement a HIRTE.

Thus, we regard the HIRTE as a framework.

6.3 Impact on the Development Process

interventions and conflicts

Full adherence to the concepts of the HIRTE – established by the requirements, constraints and patterns – implies a significant intervention in the development process in general and specifically programming. In the context of established high-integrity respectively mission-critical systems, we already find strict formalisms and standards, like in the ETC parent automotive (cmp. 2.1.1.4). These, however, often present use cases and functionality that are equally restricted: the complexity of transforming systems, with a stateless relation between input and output (cmp. 2.1.1.1), lies e.g. in numerical computation algorithms rather than the management of interactive services and transactions handling heterogenous information entities from sensor data to CDRs.

reliability vs. costs (again)

When a project necessitates the unification of different, or even conflictive development techniques (as discussed in 2.4), it seeks to strike a similar balance in its practical methods to what we were confronted with in the architecture concept. In the given case of the mission-critical telematics system, attempting to apply the full suite of formal verification methods established in high-integrity systems (cmp. e.g. [Ba03]) is very likely (deduced from industrial experience) to exceed acceptable economic limits to costs.

implications of the HIRTE

A solid compromise is presented with this work: a framework regulating the design and anticipating crucial parts of the programming ensures the safety of a significant portion of the system by standard reuse of validated components – reliability by construction. However, if not consistently perpetuated, the induced reliability is compromised. For the development of HIRTE service components this has the following implications:

Design as state automatons – this work is not the first that propagates a state-based software structure and flow of control. [Sa02] and [WSWW06] make cases for the application of state machines in the design and implementation of software systems. However, even with the domain-specific, concrete reference frame given here, a development team still needs to understand and translate component behavior in a meaningful and efficient way. And while the formal definition and introduction (syntax, semantics) of any state automaton approach may be communicated mechanistically in the preparation of a project, a standard or even realizable recommendations for corresponding "good design" is more elusive (also cmp. 6.4).

Prudent actions, internal and external operations – in 5.1.4, we provided a preliminary guideline to the selection and definition of actions and operations as specified for the HIRTE. Especially the external form is prone to misuse, if its scope and complexity of program flow are not clearly specified. Disciplined addition and implementation of actions (SRSM) and external operations (VCU) has to avoid excessively delegating business logics from the automaton to procedures, therefore depreciating controllability.

Statically resolvable structures – deny accustomed techniques and data structures, demanding alternative methods or realizations. This forces the developer in both design and implementation to think ahead to the intended deployment and allowed parameter ranges for buffers etc., as these may not be resized during run-time. Also, it influences the design and instantiation of class hierarchies, as e.g. overloading and dispatching are restricted.

Employment of a safe language or subset and RTE – in principle, any language may implement the HIRTE. As long as the development process involves manual programming or, as a matter of fact, modeling, that is not submitted to exhaustive automatic correctness checks, we strongly suggest the Ada subset described and justified in 3.2.2. Alternatively, the MISRA subsets of C/C++ might be considered (cmp. [MI04]). Regarding the RTE, any nondeterministic features like garbage collection should be avoided – or deactivated due to the static implementation – at all costs, as it may frustrate any attempt on reproducing failures related to memory management[3].

the harsh reality

Sadly, we are also well aware that Ada today has become a rather "academic" recommendation, at least outside of the specialized domains of e.g. defense and avionics. Due to the alleged liberation from the chores of memory and pointer management (that actually may help to gain a deeper understanding of programming and software behavior), highly dynamic languages like Java are now prevalent; first in education, now in the IT workforce. Arguably, it is easier for an experienced developer to migrate from Ada, C or C++ to Java than vice versa. Especially in larger companies, it is our experience that we are now at a point where high-integrity experts and senior programmers proficient in more sophisticated languages, or programming techniques beyond the basic adoption of standard libraries, are crowded out and considered too expensive. With regard to the effective omnipresence of embedded software and its collateral strategic economic significance, this is a problematic trend, unlikely to be solved by forcing unsuitable concepts on sensitive products.

6.4 State-forming Heuristics

states in general

Defining a state formally is a common enough exercise: algebraically (cmp. e.g. 4.1.3) as well as a metamodel element, e.g. of UML ([OMG092], 15.3.11), we find precise and serviceable definitions. Coming from a clear syntax, the generic (run-time) semantics quickly become more elusive with every step away from a strictly mathematical model – compare for example the contrast between the concise execution transition relation $\rightarrow_G$ on Petri nets (cmp. [CL08], 4) and "a situation during which some (usually implicit) invariant condition

[3]Remember that an OBE application does not necessarily control the full OBE; other components beyond HIRTE might not abandon dynamic memory management. Keeping the HIRTE memory consumption static retains it as the monitoring instance unlikely to be the cause of such a problem, and – as intended – a valuable facility for rapid troubleshooting.

holds" with 14 pages descriptions of states, state types, intended behavior, semantics and examples for the UML.

application states

And what then makes a proper state in the context of an application? How does a system architect, a software designer derive actual and meaningful states for the module structure from the specification of the business logics? We do not presume to be able to provide a universal answer to these questions (or, as a matter of fact, that such an attempt would even be sensible). Instead, a valid compromise might be found in domain-specific heuristics, i.e. some scheme to capture the states of a service component in an ETC OBE software.

potential indicators

A systematic approach to gain such a methodical description could consider the following indications for the information available and related to component state synthesis. Reflecting three views on associated state set abstractions,

- the *application program state* of a service[4] composing the variables, in this case the task attribute sets, the message queues and – yet lacking a program counter – a step in the business process (2.3.8) or corresponding aspect-related activities (2.3),
- the *infrastructure processing state*, e.g. given by a VCU interpreter, in analogy to the application-independent states of a POSIX thread ([IO08]), and
- *fault states* (ref. 3.1.1) as results of a risk analysis, potentially raised or detectable by the software

should provide some orientation respectively purchase to derive and form a concrete state set.

granularity

The granularity of the resulting set in relation to the program logics is a further crucial parameter for examination. On one end of the spectrum, states immediately derived from the program would map to a new configuration in the state tracer (cmp. 4.2.2) on every variable update – actually a feasible approach for specific values, think of current toll atom identifications, but hardly generalizable. On the other hand, keeping the application states on a level of abstraction with infrastructure situations, e.g. processing action, sending message etc., would be too vague to gain an understanding of the application's behavior during operations.

With viable indicators for state derivation pinpointed, the compilation of an actual heuristic lies out of the scope of this work and is left for future research.

[4]We assume that the component structure is determined prior to the behavior, based on the use cases, scenarios and superordinate business activities.

6.5 Formal Validation and Certification

helpful artifacts

The process of devising the HIRTE created, among others, three kinds of artifacts:

1. *a model* specifying the intended structure and behavior of the system,
2. *source code* implementing the model and realizing the specified behavior and
3. *traces* of the measured behavior in specific scenarios, respectively output of the system corresponding to defined calls and messages.

The quality of each artifact notwithstanding, these represent a sufficient basis for different methods of validation.

model checking

Provided a sufficient formality that allows processing of the model structures by tools, they may check it for a range of properties (cmp. e.g. [Bé01]). A rewarding and multifaceted subject is *consistency* (cmp. e.g. [Ste032]), allowing assessment of structural as well as behavioral traits including the semantic level. Checking for discrepancies and contradictions e.g. between operations declared for a component and the corresponding actions in its statechart, or between statechart message input/output and associated sequence diagrams with their message orders serves to identify problems at an early phase.

systems validation languages

Aside from the model checking features of the used tool, transforming the model into process algebra expressions respectively a systems specification language yields a complementing artifact – and view on the software and its architecture – for examination. Aside from said consistency checks, tools like FDR2 ([Ro98]) for CSP or SPIN ([Ho03]) for PROMELA allow simulation of the processes, and thusly conclusions about service behaviour: life-/deadlocks, reachability of states, fairness of scheduling and occurence of problematic state configurations. In the case of the HIRTE, such a transformation is facilitated by the step semantics defined by the automaton of 4.2.3.2 (cmp. e.g. [Hi99] for an example of step semantics in CSP). It provides an unambiguous behavioural specification that can be directly employed in the translation, detailing a service's dynamic properties for the verification.

HIRTE model transformation

Our intended business process for ISMC program development already considers an appropriate model transformation (cmp. 4.2.3.2, fig. 4.13). Especially for the implementation of service components as VCUs, code generation based on a model is a reasonable approach, making sure that a communicable specification is maintained and – once the compilation itself is stable – helping lower errors. Consequently, the formalism necessary to map the model to ISMC, its detail, completeness and unambiguousness, is also a workable basis for a mapping to a verification language.

static source code verification

Furthermore, the source code of our reference implementation goes a long way towards permitting formal static validation, excluding many language features considered unsafe or hard to determine in their run-time behaviour. Further restriction of access types respectively pointers (in the case of the VCU code, they are negligible, anyway) would enable us to extend the Ada programs e.g. with SPARK ([Ba03]) annotations, effectively eliminating constraint, storage, program and tasking errors (cmp. [Ba03], 3.1).

traces and log querying

A beneficial implication of the state machine structures of the HIRTE and its component services is that the implementation systematically produces uniform state and message traces, i.e. a component is supposed to log its current state in a defined format and prescribed container. Provided equal names or a mapping, this makes them comparable with the corresponding elements of the model. According queries of the state tracer log thus facilitate alignment and adjustments between the implementation and

- the statecharts defined for each component, describing its service behaviour (filtering the log for the entries of a specific component ID),
- inter-component activities detailing the business processes of the application (filtering for a subset or the set of all components) and
- interactions between components or the application and external systems (filtering for message queue access).

certification

In the domain of mission-critical telematics, *certification* is an important and recurring topic. Whether it aims at allowing installation of a component into an automotive environment, e.g. connection to a bus, or the participation of an OBE software in an ETS, the provider respectively the instance held liable for proper operations has to establish a defined, approved process to reach acceptance. With appropriate and reproducible methods, this facility needs to determine the degree of qualification of a new element for accurate interaction and integration with the operator's system.

certifying HIRTE conformance

Conveniently, for a software component, a certification procedure may re-utilize artifacts of the verification:

- The *specification*, in the case of HIRTE the patterns, provides a component supplier with the static and dynamic requirements to implement against. It details what is expected from a product to be introduced into the corresponding system environment.
- A *reference implementation* may either support the component supplier in the role of a framework (cmp. above 6.2), or, unpublished, serve as benchmarking resource for trial runs in an acceptance procedure. As a compromise, a subset of the reference modules may realize drivers and stubs during the development of a conforming product, allowing the successive integration of new modules. For example, it may provide a temporary ISMC interpreter for early tests of services that is later replaced

by a proprietary implementation (in another language, in optimized form etc.).

- *Traces* produced by both the reference system and the qualified product measure demanded and achieved qualities. Aside from expected traces on specified stimuli, timestamps of state tracer entries report on the fulfillment of real-time constraints. Additional threshold values (e.g. response times of OTA transactions) and expected output (e.g. format and content of receipt record messages) may be introduced by center and other infrastructure elements of the overall deployment.

6.6 Venturesome and Bolder Research Notions

6.6.1 Dynamic Fragment Generation

The DSMF mechanism introduced in 4.2.4 supports a use case that delegates a declared part of an implemented state automaton to a remote system for execution. This delegation was motivated mainly by performance, security and handling issues (e.g. interoperability efficiency) and confined to the extraction of a static, defined segment of an ISMC program.

Intuitively, it would violate our carefully established requirement for static implementation, but the question for rewarding use cases that could benefit from the dynamic construction of machine fragments, arises. Whether to serve a specific type of remote infrastructure or mode of transmission, the machine could specify a template, parameters or other complementable concept that the interpreter may use to generate the specialized, customized fragment.

6.6.2 State Machine Fragments and Homomorphic Encryption

During the work on this thesis, [Ge10] published some significant progress towards the applicability of homomorphic encryption. Being able to perform meaningful operations on ciphertexts without compromising security would enable the outsourcing of confidential computations.

An important aspect of the DSMF application in 5.3 was the operator's ability to contain not only private keys and other security-relevant elements on the smart card respectively SAM, but also its proprietary algorithms, only selectively having to distribute them. Actual, efficient future solutions for homomorphic encryption could extend this use case to scenarios with higher security requirements, completely securing the state automaton while still retaining distributability of fragments.

This raises the question if it would be possible to map the available operations of (fully) homomorphic cryptosystems to the – or at least some – step operations of an ISMC interpreter.

6.7 Tools Applied to This Work

The UML models were designed with *Enterprise Architect* up to *Version 7.5*, © SparxSystems 1998-2009.

The Ada reference sources were written and tested with the *GNAT Programming Studio (GPS), Version 4.3.1, GPL Edition* (all produced sources are published with this work), © AdaCore 2001-2008.

The document itself was written in LaTeX using *MiKTeX* up to *Version 2.8* and the *TeXnicCenter, Version 1 Beta 7.01*, © TeXnicCenter.org 1999-2006. The English is supposed to be inclined toward the American conception.

Additional diagrams were done with *Microsoft Office PowerPoint 2007*, © Microsoft.

Acknowledgements

This work is dedicated solely to my wife Sandra, who had to put up with a grumpy hermit for years and supported me indispensably. I love you with all my heart, more than anything else in this world and beyond.

I am very grateful to Prof. Dr. Franz Josef Rammig for accepting me as a doctoral candidate at an advanced stage (referring to both the thesis and my age :-), helpful reviews and support.

Many thanks (late, but anyway!) to Klaus Füller and Eckhard Müller of the Georg-Christoph-Lichtenberg-Schule in Kassel for inciting my enthusiasm for informatics beyond C64 and Amiga geekdom and imparting knowledge I still benefit from today.

To Arjen Klei, owner of omp computer gmbh, my gratitude for many years of constructive discussions and invaluable shared experience in (sometimes a bit too exciting) projects. Of my colleagues at omp, especially Thomas Herbrüggen was a great help in saving me from a lot of micromanagement. Michael Gollan provided valuable input concerning comprehensibility and certain communications engineering issues.

Jan Stehr, November 2010

Beckmesser	*den Vorhang aufreißend:*
	Seid Ihr nun fertig?
Walther	Wie fraget Ihr?
Beckmesser	Mit der Tafel ward ich fertig schier.
	Er hält die ganz mit Kreidestrichen bedeckte Tafel heraus;
	die Meister brechen in ein Gelächter aus.

Glossary and Abbreviations

Note: this glossary does not intend to give universally valid explanations. It provides definitions explicitly for the specific context of this work. Thus, certain descriptions may intentionally differ from other sources.

ABS
: Anti-lock Braking System. Sensors detect locking brakes, a controller adjusts accordingly to ensure control of the vehicle.

ACC
: Adaptive Cruise Control. Integrates sensor feedback data from its environment, e.g. other traffic as measured by radar.

Access Type
: A fat pointer in Ada. In contrast to the thin memory *address*, it includes additional information, e.g. constraints (size, first, last).

Activity
: Program/control flow, events and transactions implementing the use cases and facilitating *business processes* of a system. May be partitioned over a set of executing components respectively *processes*.

Address
: Memory address; a thin pointer in Ada in contrast to the *access type*.

APDU
: Application Protocol Data Unit. Carries application data between a smart card and terminal.

API
: Application Programming Interface. Commonly encapsuling and providing access to some function/class library, framework or device.

App
: Common, established term for smartphone applications.

ASN.1
: Abstract Syntax Notation One. Language to describe data structures for

the purposes of encoding, transmitting and decoding.

ATAM
Architecture Tradeoff Analysis Method. A heuristic approach to evaluate the qualities of software architectures.

AUTOSAR
Automotive Open System Architecture. One standard for automotive software with a focus on scalability and maintainability.

Billing
Compiling *priced/rated* services and presenting it to a debitor.

Black Box
A component of which only the inputs and outputs, i.e. externally measurable behavior are/is considered.

BS 26
Bearer Service 26. Allows transmission of data in a *GSM* network with a maximum bandwidth of 9,600 bit/s.

Business Logics
See *activity*, but generally limited to the behaviour of a single component.

Business Process
A specified commercial activity implemented by an organization, e.g. a company.

CAN
Controller-Area Network. An event-driven automotive bus standard for the communication between *ECU*s, up to 1 Mbit/s.

CDR
Call Detail Record. A record describing attributes of a phonecall (or other similar services) relevant for *billing* in a *CN*.

Charging
The process of debiting an account with the amount associated with a priced service or unit, e.g. a toll atom.

CN
Cellular Network. Any network for cellular mobile communications, based on e.g. *GSM* or *UMTS*.

Compile Time
Interval of the translation of software code into an executable in a single pass or multiple passes. Note that in the context of statically determinable code, there is a distinction between checks by the compiler and checks at compile-time that are not necessarily performed by the compiler itself, but by any other tool prior to execution or *run-time*.

Component
A module of a system with defined interfaces to its environment or other components. Also a complete (sub-)system in the context of large-scale architectures. May be composed of other components.

CSP
Communicating Sequential Processes. A process algebra for the description of concurrent processes interacting via messages.

Dead Reckoning
Estimating positions based on the last valid fix and sensors still available, e.g. compass, gyroscope and/or tachometer, after loosing a *GNSS* signal.

Dispatching
Run-time determination of method code to be called.

DSMF
Distributable State Machine Fragment. A composite state that is self-contained, i.e. its attributes enable interpretation and execution outside of its parent machine.

DSRC
Dedicated Short Range Communications. Infrared or microwave band data transmissions between *OBE* and roadside infrastructure.

Dynamic Binding
Determination of the implementation of an accessed operation or object during *run-time*.

eCall
Automatic emergency call. On sensors detecting a crash, the vehicle's position is transmitted to a rescue dispatcher.

EBNF
Extended Backus-Naur Form. Metasyntax notation for the definition of context-free grammars. Standardized in ISO/IEC 14977.

ECU
Electronic Control Unit. Programmable microcontroller to implement vehicle functions in software.

EETS
European Electronic Toll Service. Umbrella term for activities and standards of the EU to establish an interoperable tolling infrastructure.

EETS Provider
Operator responsible for the technical implementation and management of an *EETS* system.

EEV

Enhanced Environmentally-friendly Vehicle. European emission standard for the definition of (comparably) clean vehicles.

EFC
Electronic Fee Collection. See *ETC*.

EEPROM
Electrically Erasable Programmable Read-Only Memory. Non-volatile memory, persistent data after power-off.

ESC
Electronic Stability Control. Intervenes in braking and acceleration to prevent skids.

ETC
Electronic Toll Collection. Means to (semi-)automatically levy toll for traffic infrastructure usage.

ETC Software
Implements the *business logics* of distributed *ETC OBE*.

ETS
Electronic Toll Service. Used synonymously for *ETC*.

Execution Environment
Hardware and/or software resources to run software programs on, e.g. a microcontroller or an operating system. Ranges from basic, like a processor, memory, I/O, to concurrent thread and process management and communication busses.

Execution Unit
A generic term for *process* and *thread*.

FCD
Floating Car Data. The concept of collecting position, direction and speed of vehicles e.g. for traffic analysis and control.

Flash
Non-volatile memory based on *EEPROM*.

FlexRay
A time- and event-triggered automotive bus protocol for up to 10 Mbit/s.

FMS
Fleet Management Standard. Allows telemetry *CAN* readout of vehicle data, e.g. current engine parameter values, fuel consumption.

FSM
Finite State Machine.

Galileo

A civillian *GNSS*, issued by the European Union, if installed would be based on 30 satellites.

GNSS
Global Navigation Satellite System. Any satellite constellation allowing positioning by receiving, decoding and evaluating its signals.

GPRS
General Packet Radio Service. Provides data transfer in *GSM* networks, approx. 40 kbit/s in practical application.

GPS
Global Positioning System. A *GNSS*, operated by the US military, consisting of 24 to 32 satellites.

GSM
Global System for Mobile communications. During the time of this work, the globally most accepted standard for mobile phones.

HI
High Integrity. A software quality aiming at verifiable, systematic stability and reliability. "Integrity" in the context of a program generally refers to (the absence of) constructs considered unsafe in the respective language, e.g. dynamic memory management and binding.

HIS
Herstellerinitiative Software. One of the standard frameworks for automotive software modules.

HMI
Human Machine Interface. Any user interface, implemented in hardware or software.

Homomorphic Encryption
Allows meaningful operations on cyphertexts without the need for prior decryption or keys. Realizes programs that operate on encrypted input to produce an encryption of their output without the need to decode the input during execution.

Informing System
Component providing convenient, non-critical data to the user or environment, e.g. navigation solutions, news, entertainment. Failure has no impact on physical integrity of the user, others, or significant commercial transactions.

Interactive System
A system or component that during its execution processes transactions, exchanges messages, generally interacts with the user or other components to produce results.

ISAM
Indexed Sequential Access Method. A database organization approach with the index implementation based on B-trees.

ISMC
Interpretable State Machine Code. Designates either the language for representing state machines as a sequence of *opcodes* conforming to the specification introduced by this work, or a corresponding program.

ISO
International Organization for Standardization.

ITU
International Telecommunication Union.

MDn
Message Digest algorithm (*n* denoting the version). A cryptographic hash function.

Microkernel
Component providing basic memory management, scheduling of threads of execution and inter-process communications.

Mission-Critical System
Component providing crucial data to the user or environment, i.e. failure could lead to physical harm or cause substantial monetary losses.

MSISDN
Mobile Subscriber Integrated Services Digital Network Number. Uniquely identifies a subscriber in a *GSM* or *UMTS CN*.

MTBF
Mean Time Between Failures. The average time between failures of a system.

Mutex
Mutual exclusion. Algorithms and structures to control concurrent access, e.g. rendezvous, semaphores, monitors.

OBE
On Board Equipment. Electronic components distributed in a vehicle, connected by some bus system. The aggregate provides an *execution environment* for software.

OBE Software
Any component running on the *OBE*.

OBU
On Board Unit. An integrated *OBE* solution. The intended use cases determine the sensors, interfaces and components of the OBU.

OBU Software
Any component running on the *OBU*.

OMS
OBE Management Software. Broker between an *ETC* application implemented on a smart card and its required *OBE* resources.

Opcode
Operation Code. Machine language (binary) specification of an operation to perform by the RTE.

OS
Operating System. A common *RTE* for applications.

OSEK/VDX
Offene Systeme für die Elektronik im Kraftfahrzeug / Vehicle Distributed eXecutive. A standard framework for automotive software modules, providing an ISO/ANSI-C interface.

OSI
Open Systems Interconnection. An *ISO* standard that describes a reference model for communications and protocols with seven layers (Application, Presentation, Session, Transport, Network, Data-Link, Physical; top to bottom).

OTA
Over The Air. Interfaces, protocols and interactions utilizing a *CN* for communications between system components.

PC/SC
Personal Computer/Smart Card. Specification for *smart card* system integration. Unlike the name might suggest, implementations are available for a wide range of platforms, including smartphone *OSs*.

PER
Packed Encoding Rules. *ASN.1* definition for encoding data units using the minimum number of bits.

POSIX
Portable Operating System Interface (for Unix). A set of standards defining software interfaces of a Unix *OS*.

Pricing
The process of associating a service or unit, e.g. a toll atom, with an amount of money in conformance to a defined *tariff scheme*.

Process
Any structured sequence of activities, steps, actions or commands. Technical view: a program execution with its own memory address space.

May aggregate and manage sub-processes in the form of *threads*. For the business view see *Business Process*.

Program Counter
Represents respectively stores the position of the current instruction during program execution.

PROMELA
Process or Protocol Meta Language. A language for the modeling and verification of concurrent processes interacting via message channels.

Provisioning
The process of outfitting and maintaining a system to enable it to provide services, respectively a component to enable its participation in a system.

QoS
Quality of Service. Any measurement that allows assessment of the performance or output of a system respectively its processes.

RAM
Random Access Memory. In the context of embedded systems, this is a limited resource, which is required to adhere to strict specifications concerning reliability, e.g. operation temperature ranges, timing and voltage fluctuation.

Rating
Very close or equal to *Pricing*.

RCI
Road Charging Interoperability. Predecessor to *EETS*.

Reactive System
A component continually interacting with its environment.

Refurbishment
Overhauling and updating of used hardware components for the purpose of reissue.

Roaming
A mobile phone using another *CN* than its home. The visited network, commonly in another geographical area, provides services under special conditions concerning available features and billing.

ROM
Read Only Memory. See *RAM* for context.

RPC
Remote Procedure Call. Invoking a procedure in another *RTE*, involving some protocol to specify name, parameters and retrieve return values.

RTE
Run-time Engine or Run-time (Execution) Environment. A platform which can execute a software program. May consist of hardware (processor, memory, I/O interfaces) and software (scheduling, memory management, process communication), designate a software solution running on unspecified hardware or refer to a formal machine specification.

Run-time
Interval of a program's execution.

Run-to-completion
An execution model of e.g. a program or *ISMC* that assumes that specific operations like event processing or procedure invocation have to be completed before processing of the next operation can begin. While avoiding the need for concurrency mechanisms in the executable, it requires event/message queues to handle requests received during these atomic activities.

SAM
Secure Access Module. Encapsulates cryptographic algorithms, often implemented on a smart card.

Scanner
A lexical analyzer that compiles a sequence of bytes and converts it to a token of a specific type.

SHA
Secure Hash Algorithm. A cryptographic hash function.

SIL
Safety Integrity Level. A relative level of risk reduction measured by various quantitative and qualitative characteristics. Ranges from SIL1 (least dependable) to SIL4 (most dependable).

SIM
Subscriber Identity Module. Commonly a smart card that securely stores subscriber-specific data and keys for a *CN* provider in a mobile phone.

Smart Card
8 or 16 bit processor with 6 to 64 kb of memory and I/O encapsuled in a smart card module. *OS* is stored in *ROM*, memory data access from the outside is controlled by the processor and its software.

SMS
Short Message Service. Provides interchange of messages up to 140 bytes in a *GSM CN*.

Spec
Specification. In the context of Ada, a specification file, defining data

structures and operation signatures.

SPOF
Single Point of Failure. System element, whose failure implies failure of the entire system.

SQL
Structured Query Language. Language for the management, querying and modifying data of relational databases.

SRSM
Statically Resolvable State Machine. Refers to a pattern introduced by this work. A state automaton configuration respectively structure that does not rely on polymorphy, dispatching or similar dynamic concepts during run-time.

Step Semantics
of a state machine define a decomposed execution of its structure with a starting point, conditioned processing of states, transitions, (where applicable parts of) their actions, events and signals, and a completion point from which the starting point of the successive step can be derived, if the machine did not reach its final state.

T=0
Asynchronous, byte-oriented, half-duplex transmission protocol between a terminal and *smart card*. Simple, requires little memory.

T=1
Asynchronous, block-oriented, half-duplex transmission protocol between a terminal and *smart card*. Allows secure messaging.

Tariff Scheme
Defines the price of a service or unit dependent on a set of parameter values, e.g. time, user attributes.

Telematics
Fusion of technologies, methods and processes of both telecommunications and informatics. Prominent application examples are traffic, healthcare and defense.

Thick Client
OBE outfitted with sufficient resources (*execution environment*) to completely process tolling data and provide a result to send to a center component for *billing*.

Thin Client
OBE with minimum run-time resources that merely collects and transmits tolling-relevant data, leaving complex computations to center systems.

Thread
Sequential execution of commands running in the context (memory address space, management) of a *process*.

TLV
Type or Tag Length Value. Encoding of message data units according to the first bytes representing the type of the unit's value, the next bytes representing the length n of the value, followed by the n bytes of the value itself.

Toll Charger
Organization determining the business and legal schemes, processes and parameters of an *EETS* system. Responsible for the collection and cession of tolled amounts to the government (where applicable).

Tolling Scheme
See *Tariff Scheme*.

Transforming System
Component implementing a stateless, functional relationship between the input and output values. Takes input and deterministically converts it into a specified result.

Turing Machine
A theoretical computation model consisting of a symbol tape, head, registers and a transition function.

UMTS
Universal Mobile Telecommunications System. *CN* standard technology allowing packet-oriented uplink and downlink between approx. 384 kbit/s and 7.2 Mbit/s.

VAS
Value Added Service. Utilizes the infrastructure of some base service, e.g. *ETC*, to offer supplemental functionality, e.g. vehicle tracking.

VCU
Virtual Control Unit. Run-time interpreter for *ISMC*, with associated message queues and function sets for interaction and action implementation.

VFB
AUTOSAR Virtual Function Bus. Interconnection middleware abstracting from the underlying automotive hardware, e.g. the busses.

Virtualization
The concept of emulating a system component that realizes a *RTE* for other components, e.g. a whole computer for running an OS, or a JavaVM for bytecode programs.

White Box
A component of which both the externally measurable behavior and internal states are considered.

WLAN
Wireless Local Area Network. Connects two or more devices, providing approx. 11 Mbit/s.

Bibliography

[3G03] 3rd Generation Partnership Project: 3GPP TS 51.014 V4.3.0, Technical Specification Group Terminals; Specification of the SIM Application Toolkit for the Subscriber Identity Module – Mobile Equipment (SIM – ME) interface (Release 4). 3GPP, 2003.

[Ao06] Aonix, Inc.: ObjectAda Real-Time RAVEN Product Fact Sheet. www.aonix.com, 2006.

[AU06] AUTOSAR Administration: Specification of RTE Software, Version 1.0.1. AUTOSAR GbR, 2006.

[AU062] AUTOSAR Administration: Specification of Module FlexRay Interface, Version 2.0.0. AUTOSAR GbR, 2006.

[AU063] AUTOSAR Administration: Requirements on RTE, Version 1.0.1. AUTOSAR GbR, 2006.

[Ba03] J. Barnes: High Integrity Software. Addison Wesley, 2003.

[BA04] K. Beck, C. Andres: Extreme Programming Explained, 2nd Edition. Addison Wesley, 2004.

[BCK03] L. Bass, P. Clements, R. Kazman: Software Architecture in Practice, Second Edition. Addison Wesley, 2003.

[BDV03] Burns, Dobbing, Vardanega: Guide for the use of the Ada Ravenscar Profile in high integrity systems. University of York technical report YCS 348, 2003.

[Be06] M. Ben-Ari: Principles of Concurrent and Distributed Programming, Second Edition. Addison Wesley, 2006.

[Bé01] B. Bérard et al.: Systems and Software Verification: Model-Checking Techniques and Tools. Springer, 2001.

[BM98] E. L. A. Baniassad, G. C. Murphy: Conceptual Module Querying for Software Reengineering. Proceedings of the 20th International Conference on Software Engineering, 1998.

[BMJ05] Bundesministerium der Justiz: Gesetz über den Betrieb elektronischer Mautsysteme (Mautsystemgesetz – MautSysG). www.juris.de, 2005.

[BMVBS08] Bundesministerium für Verkehr, Bau und Stadtentwicklung: Die Lkw-Maut: Fragen und Antworten. www.bmvbs.de, 2008.

[Bo81] B. W. Boehm: Software Engineering Economics. Prentice Hall, 1981.

[Bo01] Robert Bosch GmbH (Ed.): Mikroelektronik im Kraftfahrzeug. Robert Bosch GmbH, Stuttgart, 2001.

[Bo02] Robert Bosch GmbH (Ed.): Konventionelle und elektronische Bremssysteme. Robert Bosch GmbH, Stuttgart, 2002.

[Bo022] Robert Bosch GmbH (Ed.): Autoelektrik/Autoelektronik, Systeme und Komponenten, 4. Auflage. Vieweg Verlag, 2002.

[BT05] Becker, Timm-Giel: Selbststeuerung in der Transportlogistik: Modellierung der mobilen Kommunikation. Industrie Management, 5/2005, S. 71-74, GITO-Verlag, Berlin.

[Bu05] G. C. Buttazzo: Hard Real-Time Computing Systems, Second Edition. Springer, 2005.

[BW98] A. Burns, A. Wellings: Concurrency in Ada. Cambridge University Press, 1998.

[BW06] B. Brosgol, A. Wellings: A Comparison of Ada and Real-Time Java for Safety Critical Applications. Reliable Software Technologies – Ada Europe 2006.

[BW09] A. Burns, A. Wellings: Real-Time Systems and Programming Languages, Fourth Edition. Addison Wesley, 2009.

[C2C08] Car 2 Car Communication Consortium Website. www.car-2-car.org, 2008.

[CA08] R. Chapman, P. Amey: SPARK 95 – The SPADE Ada 95 Kernel (including RavenSPARK), Edition 4.8. Praxis High Integrity Systems, 2008.

[CGW91] Cullyer, Goodenough, Wichmann: The choice of computer languages for use in safety-critical systems. Software Eng. J., 1991.

[CKK02] P. Clements, R. Kazman, M. Klein: Evaluating Software Architectures. Addison Wesley, 2002.

[Cl03] P. Clements et al.: Documenting Software Architectures. Addison Wesley, 2003.

[CL08] C. G. Cassandras, S. Lafortune: Introduction to Discrete Event Systems, Second Edition. Springer, 2008.

[Co04] B. J. Copeland: The Essential Turing. Oxford University Press, 2004.

[DC021] DaimlerChrysler AG: System for Determining Road Tolls, WO 02/061691 A1. World Intellectual Property Organization, 2002.

[DC022] DaimlerChrysler AG: Control Method for Use in a Toll Determination System, WO 02/061690 A1. World Intellectual Property Organization, 2002.

[DC03] DaimlerChrysler AG: Europäische Patentanmeldung EP 1 335 324 A2. European Patent Office, 2003.

[DC06] DaimlerChrysler Services Mobility Management GmbH: Market Consultation "Anders Betalen voor Mobiliteit", Research Assignment 1: Total Cost of System and Organization for the KMP. Kostenmonitor Kilometerprijs, The Netherlands, 2006.

[DIN89] DIN Deutsches Institut für Normung e. V.: DIN 19250: Grundlegende Sicherheitsbetrachtungen für MSR-Schutzeinrichtungen. DIN, 1989.

[Do03] B. P. Douglass: Real-Time Design Patterns. Addison Wesley, 2003.

[Do04] B. P. Douglass: Real-Time UML, Third Edition. Addison Wesley, 2004.

[Dr06] D. Drusinsky: Modeling and Verification Using UML Statecharts. Newnes/Elsevier, 2006.

[eC06] eCall Driving Group: Recommendations of the DG eCall for the introduction of the pan-European eCall, Version 2.0. www.esafetysupport.org, 2006.

[ETSI96] European Telecommunications Standards Institute: Digital cellular telecommunications system (Phase 2+); Specification of the SIM Application Toolkit for the Subscriber Identity Module - Mobile Equipment (SIM - ME) interface (GSM 11.14), Version 5.2.0. ETSI, 1996.

[EU04] Official Journal of the European Union: Directive 2004/52/EC of the European Parliament and of the Council of 29 April 2004 on the interoperability of electronic road toll systems in the Community. European Union, 2004.

[EU042] Official Journal of the European Union: Corrigendum to Directive 2004/52/EC of the European Parliament and of the Council of 29 April 2004 on the interoperability of electronic road toll systems in the Community. European Union, 2004.

[EU09] Official Journal of the European Union: Commission Decision of 6 October 2009 on the definition of the European Electronic Toll Service and its technical elements. European Union, 2009.

[EVB01] Eberspächer, Vögel, Bettstetter: GSM Global System for Mobile Communication, Third Edition. B. G. Teubner, 2001.

[Fe04] H. Feng: DCharts, a Formalism for Modeling and Simulation Based Design of Reactive Software Systems. McGill University, Montréal, Canada, 2004.

[FECA04] R. E. Filman, T. Elrad, S. Clarke, M. Aksit: Aspect-Oriented Software Development. Addison-Wesley Longman, 2004.

[Fl05] FlexRay Consortium: FlexRay Communications System – Protocol Specification Version 2.1. www.flexray.com, 2005.

[FMS05] FMS-Standard Working Group: FMS-Standard Interface Description, Vers. 01.00. fms-standard.com, 2005.

[FSH04] Freitag, Scholz-Reiter, Herzog: Selbststeuerung logistischer Prozesse – Ein Paradigmenwechsel und seine Grenzen. Industrie Management 20 (2004) 1, S. 23-27.

[Ga06] Garmin International, Inc.: GPS 15H & 15L Technical Specifications. Garmin, USA, 2006.

[Ge10] C. Gentry: Computing Arbitrary Functions of Encrypted Data. Communications of the ACM, Vol. 53, No. 3. ACM, 2010.

[GH081] Green Hills Software, Inc.: Green Hills Minimal Ada Run-Time (GMART) Datasheet. www.ghs.com, 2008.

[GH082] Green Hills Software, Inc.: Green Hills Safe-Tasking Ada Run-Time (GSTART) Datasheet. www.ghs.com, 2008.

[GH93] H. Garavel, R.-P. Hautbois: An experience with the LOTOS formal description technique on the flight warning computer of the Airbus 330/340 aircrafts. Proc. First AMAST Int. Workshop on Real-Time Systems, Springer, 1993.

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns. Addison Wesley, 1995.

[Gi62] A. Gill: Introduction to the Theory of Finite-state Machines. McGraw-Hill, 1962.

[Go73] R. P. Goldberg: Architectural Principles for Virtual Computer Systems. Harvard University, 1973.

[HMPR04] A.R. Hevner, S.T. March, J. Park, S. Ram: Design Science in Information Systems Research. MIS Quarterly, vol. 28, 75-105. University of Minnesota, 2004.

[Hi92] D. Hildebrand: An Architectural Overview of QNX. Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures, 1992.

[Hi99] J. Hiemer: Statecharts in CSP. Schriftenreihe Forschungsergebnisse zur Informatik, Bd. 48, Verlag Dr. Kovac, 1999

[HI03] Herstellerinitiative Software: OSEK OS Extensions for Protected Applications Version 1.0. DaimlerChrysler AG, www.automotive-his.de, 2003.

[HI04] Herstellerinitiative Software: API IO Library Version 2.0.3. www.automotive-his.de, 2004.

[HMU07] J. E. Hopcroft, R. Motwani, J. D. Ullman: Introduction to Automata Theory, Languages, and Computation, 3rd Edition. Addison Wesley, 2007.

[Ho03] G. J. Holzmann: The SPIN Model Checker. Addison Wesley, 2003.

[Ho04] C. A. R. Hoare: Communicating Sequential Processes. www.usingcsp.com, 2004.

[HP98] D. Harel, M. Politi: Modeling Reactive Systems with Statecharts: The STATEMATE Approach. McGraw-Hill, 1998.

[HR02] P. Hruschka, C. Rupp: Agile Softwareentwicklung für Embedded Real-Time Systems mit der UML. Hanser, 2002.

[IEC85] IEC International Electrotechnical Commission: International Standard 812, Analysis Techniques for System Reliability, Procedures for Failure Mode and Effects Analysis. IEC, Geneva, 1985.

[IEC98] IEC International Electrotechnical Commission: IEC 61508 – Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems. IEC, 1998.

[IEC05] IEC International Electrotechnical Commission: Functional Safety and IEC 61508. IEC, Geneva, 2005.

[IO08] IEEE, The Open Group: POSIX.1-2008, IEEE Std 1003.1-2008, The Open Group Technical Standard Base Specifications, Issue 7. The IEEE and The Open Group, 2001-2008.

[ISO00] ISO International Organization for Standardization/IEC: ISO/IEC TR 15942: Guide for the use of the Ada programming language in high integrity systems. ISO/IEC, 2000.

[ISO02] ISO International Organization for Standardization: ISO 11898-4: Time Triggered CAN. ISO/IEC, 2002.

[ISO03] ISO International Organization for Standardization: ISO/TS 17573:2003 Road Transport and Traffic Telematics – Electronic Fee Collection (EFC) – Systems architecture for vehicle related transport services. ISO, 2003.

[ISO05] ISO International Organization for Standardization: ISO/IEC 7816-4:2005 Identification cards – Integrated circuit cards – Part 4: Organization, security and commands for interchange. ISO/IEC, 2005.

[ISO07] ISO International Organization for Standardization/IEC: ISO/IEC 8652:1995/Amd 1:2007. ISO/IEC, 2007.

[ISO94] ISO International Organization for Standardization: ISO 11898: Austausch digitaler Informationen; Controller Area Network (CAN) für schnellen Datenaustausch. ISO, 1994.

[ISO95] ISO International Organization for Standardization/IEC: Ada Reference Manual, ISO/IEC 8652:1995(E) with Technical Corrigendum 1 and Amendment 1. ISO/IEC, 1995, 2006.

[ISO96] ISO International Organization for Standardization/IEC: Information technology – Syntactic metalanguage – Extended BNF, ISO/IEC 14977. ISO/IEC, 1996.

[ISO97] ISO International Organization for Standardization: ISO/IEC 7816-3:1997 Identification cards – Integrated circuit cards – Part 3: Electronic signals and transmission protocols. ISO/IEC, 1997.

[ITU021] ITU International Telecommunication Union: ITU-T Recommendation X.680, Information technology — Abstract Syntax Notation One (ASN.1): Specification of basic notation. ITU, 2002.

[ITU022] ITU International Telecommunication Union: ITU-T Recommendation X.691, Information technology -– ASN.1 encoding rules: Specification of Packed Encoding Rules (PER). ITU, 2002.

[Ka03] S. H. Kan: Metrics and Models in Software Quality Engineering, Second Edition. Addison Wesley, 2003.

[KK01] M. Kersten, H. B. Keller: Die Problematik der Abbildung von UML-Modellen auf Konstrukte der Programmiersprache ADA. ADA und Software Qualität: 3. Ada Deutschland Tagung. Shaker Verlag, 2001.

[KLM03] Kaiser, Liggesmeyer, Mäckel: A new component concept for fault trees. Proc. 8th Australian workshop on Safety critical systems and software (SCS '03). Australian Computer Society, 2003.

[Ko97] H. Kopetz: Real-Time Systems. Kluwer, 1997.

[KPRR91] Klöppel, Paul, Rauch, Ruhland: Compilerbau. Vogel, 1991.

[Ku05] R. Kurki-Suonio: A Practical Theory of Reactive Systems. Springer, 2005.

[KWK02] Kwon, Wellings, King: Ravenscar-Java: A high integrity profile for real-time Java. Proceedings of Joint ACM Java ISCOPE Conference 2002.

[Li00] J. W. S. Liu: Real-time Systems. Prentice Hall, 2000.

[LI06] LIN Consortium: LIN Specification Package Revision 2.1. www.lin-subbus.org, 2006.

[Ma07] Mathworks Website, www.mathworks.com, 2007.

[MB02] S. J. Mellor, M. J. Balcer: Executable UML. Addison Wesley, 2002.

[MI04] Motor Industry Software Reliability Association: MISRA-C: 2004 Guidelines for the use of the C language in critical systems, 2004.

[Mi07] Microsoft Corporation: Windows Automotive Data Sheet. www.microsoft.com/windowsautomotive, 2007.

[MNS95] G. C. Murphy, D. Notkin, K. Sullivan: Software Reflexion Models: Bridging the Gap between Source and High-Level Models. IEEE CS Transactions on Software Engineering, 1995.

[MO06] MOST Cooperation: MOST Specification Rev. 2.5. www.mostcooperation.com, 2006.

[MS07] C. Marscholik, P. Subke: Datenkommunikation im Automobil – Grundlagen, Bussysteme, Protokolle und Anwendungen. Hütig, 2007.

[MVW05] Ministerie van Verkeer en Waterstaat: Anders Betalen voor Mobiliteit, Requirements Specification (draft version 0.2). The Netherlands, 2005.

[MVW08] Ministerie van Verkeer en Waterstaat: Kilometre Pricing in the Netherlands (KMP), Information Update. The Netherlands, 2008.

[OMG05] Object Management Group: UML Profile for Schedulability, Performance, and Time Specification, Version 1.1. www.omg.org, 2005.

[OMG07] Object Management Group: MOF 2.0/XMI Mapping, Version 2.1.1. www.omg.org, 2007.

[OMG091] Object Management Group: OMG Unified Modeling Language (OMG UML), Infrastructure, V2.2. www.omg.org, 2009.

[OMG092] Object Management Group: OMG Unified Modeling Language (OMG UML), Superstructure, V2.2. www.omg.org, 2009.

[OMG09] Object Management Group: UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, Version 1.0. www.omg.org, 2009.

[OS04] OSEK Group: OSEK/VDX Communication Version 3.0.3. www.osek-vdx.org, 2004.

[OS042] OSEK Group: OSEK/VDX Binding Specification Version 1.4.2. www.osek-vdx.org, 2004.

[OS05] OSEK Group: OSEK/VDX Operating System Version 2.2.3. www.osek-vdx.org, 2005.

[PB06] A. T. W. Pickford, P. T. Blythe: Road User Charging and Electronic Toll Collection. Artech House, 2006.

[PC05] PC/SC Workgroup: Interoperability Specification for ICCs and Personal Computer Systems, Part 1. Introduction and Architecture Overview, Revision 2.01.01. www.pcscworkgroup.com, 2005.

[Pl04] G. D. Plotkin: A Structural Approach to Operational Semantics (Preprint submitted to Journal of Logic and Algebraic Programming). University of Edinburgh, 2004.

[Pu01] de la Puente et al.: Open Ravenscar Real-Time Kernel Operation Manual Version 2.2b. Universidad Politécnica de Madrid, 2001.

[Ra06] W. Rankl: Chipkartenanwendungen – Entwurfsmuster für Einsatz und Programmierung von Chipkarten. Hanser, 2006.

[RCI07] Road Charging Interoperability Project Consortium: Minimum Architecture for Interoperability, Version 1.01. ERTICO ITS Europe, 2007.

[RCI072] Road Charging Interoperability Project Consortium: Consortium high-level view on RCI architecture and specifications, Version 1.1. ERTICO ITS Europe, 2007.

[RE08] W. Rankl, W. Effing: Handbuch der Chipkarten, 5. Auflage. Hanser, 2008.

[RE92] RTCA/EUROCAE: Software Considerations in Airborne Systems and Equipment Certification. RTCA/DO-178B; EUROCAE/ED12-B. RTCA, EUROCAE, 1992.

[RG06] K. Rüdiger, M. Gersch: In-Vehicle M-Commerce: Business Models for Navigation Systems and Location-based Services. Embedded Security in Cars. Springer, 2006.

[Ri00] D. Riehle: Framework Design. A Role Modeling Approach. Swiss Federal Institute of Technology, Zurich, 2000.

[Ro98] A. W. Roscoe: The Theory and Practice Of Concurrency. Prentice Hall, 1998.

[RW05] P. Robertson, B. Williams: A Model-Based System Supporting Automatic Self-Regeneration of Critical Software. Proceedings SelfMan, 2005.

[Sa00] B. I. Sandén: Implementation of state machines with tasks and protected objects. Ada User Journal 20:4, 2000.

[Sa02] M. Samek: Practical Statecharts in C/C++. CMP Books, 2002.

[Sc07] H. Schwichtenberg: Bestandsaufnahme – Programmiersprachen in Lehre und Praxis. Heise, iX 06/2007.

[Sc96] B. Schneier: Applied Cryptography, Second Edition. Wiley, 1996.

[Si06] Siemens AG: Anders Betalen Voor Mobiliteit Phase 2 Market Consultation, Total Cost of System and Organization for KMP, Siemens response. Siemens, 2006.

[SKG08] Special Knowledge Group "Anders Betalen voor Mobiliteit": Report round table discussions SKG ABvM September 2008 "Collection and (forced) debt collection (NL: inning en dwanginvordering)", Version 1.0. ITS Netherlands, connekt, 2008.

[SRM06] Stephan, Richter, Müller: Aspects of Vehicle Software Flashing. Embedded Security in Cars. Springer, 2006.

[SS10] Sparx Systems Ltd. Website. www.sparxsystems.eu, 2010.

[ST06] SPARK Team: The SPARK Ravenscar Profile, Issue 1.5. Praxis High Integrity Systems, 2006.

[Ste03] J. Stehr: Aspects of Mobile Telematics Applications and Services. European Journal of Navigation, 2003.

[Ste032] J. Stehr: Semantische Konsistenzprüfung von UML-Verhaltensdiagrammen zur Modellierung von eingebetteten Systemen. Ein formaler Ansatz durch Abbildung in die Prozessalgebra CSP. Universität Paderborn, 2003.

[Ste06] J. Stehr: Modellgetriebene Entwicklung von Mautsystemen: Domänenspezifische Aspekte der Verkehrstelematik. ObjektSPEKTRUM 04/2006.

[Sto96] N. Storey: Safety-Critical Computer Systems. Prentice Hall, 1996.

[Sun061] Sun Microsystems, Inc.: Runtime Environment Specification, Java Card^{TM}Platform, Version 2.2.2. www.sun.com, 2006.

[Sun062] Sun Microsystems, Inc.: Virtual Machine Specification, Java Card^{TM}Platform, Version 2.2.2. www.sun.com, 2006.

[Sun063] Sun Microsystems, Inc.: Application Programming Interface, Java Card^{TM}Platform, Version 2.2.2. www.sun.com, 2006.

[SZ06] J. Schäuffele, T. Zurawka: Automotive Software Engineering, 3. Auflage. Vieweg, 2006.

[Ta97] A. S. Tanenbaum: Operating Systems, Second Edition. Prentice Hall, 1997.

[TC07] Toll Collect GmbH: Lkw-Maut in Deutschland. Nutzerinformationen. www.toll-collect.de, 2007.

[Te02] T. Tempelmeier: On The Real Value Of New Paradigms. OMER — Object-oriented Modeling of Embedded Real-Time Systems. Gesellschaft für Informatik, 2002.

[VGRH81] Vesely, Goldberg, Roberts, Haasl: Fault Tree Handbook (NUREG-0492). U.S. Nuclear Regulatory Commission, 1981.

[VM04] Bundesrepublik Deutschland: V-Modell XT. www.v-modell-xt.de, 2004.

[Wa05] A. Wasowski: Code Generation and Model Driven Development for Constrained Embedded Software. IT University of Copenhagen, 2005.

[We07] T. Weilkiens: Die Wogen glätten sich. ObjektSPEKTRUM 04/2007.

[WSWW06] Wagner, Schmuki, Wagner, Wolstenholme: Modeling Software with Finite State Machines. Auerbach Publications, 2006.

[Zi80] H. Zimmermann: OSI Reference Model —- The ISO Model of Architecture for Open Systems Interconnection. IEEE Transactions on Communications, Vol. 28, No. 4, April 1980.

[ZS07] W. Zimmermann, R. Schmidgall: Bussysteme in der Fahrzeugtechnik – Protokolle und Standards. 2. Auflage. Vieweg, 2007.