TECHNISCHE UNIVERSITÄT MÜNCHEN
Lehrstuhl für Echtzeitsysteme und Robotik

Model-Based Development of Fault-Tolerant Real-Time Systems

Christian Buckl

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. Dr. h.c. Manfred Broy
Prüfer der Dissertation:	
	1. Univ.-Prof. Dr. Alois Knoll
	2. Prof. Edward A. Lee University of California, Berkeley/USA

Die Dissertation wurde am 05.05.2008 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 22.09.2008 angenommen.

Zusammenfassung

Die Entwicklung fehlertoleranter Echtzeitsysteme ist sehr komplex. Die Systeme müssen nicht nur Echtzeitbedingungen einhalten, sondern ihre Funktionalität auch unter dem Einfluss von durch die Hardware und/oder die Software bedingten Fehlern erbringen. Um in einem System Fehlertoleranz zu erlangen, muss Redundanz eingesetzt werden. Diese wird häufig durch Verwendung von mehreren Rechnern in einem verteilten System erzielt.

Um die wachsende Komplexität der Software zu beherrschen, werden in der Informatik modellbasierte Entwicklungswerkzeuge eingesetzt. Die existierenden Werkzeuge beschränken sich dabei jedoch zumeist auf die Modellierung und Generierung der Anwendungsfunktionalität. Code für nicht-funktionale Aspekte wie Fehlertoleranzmechanismen, Kommunikation oder Scheduling wird nicht generiert. Diese Aspekte sind jedoch für einen Großteil der Komplexität verantwortlich.

Diese Arbeit stellt daher ein Entwicklungswerkzeug für fehlertolerante Echtzeitsysteme vor, welches die Generierung von Code zur Umsetzung dieser nichtfunktionalen Aspekte unterstützt. Dadurch werden existierende Entwicklungswerkzeuge ideal ergänzt. Der Beitrag dieser Arbeit liegt insbesondere in der Ausarbeitung von geeigneten Modellen zur Beschreibung und Codegenerierung fehlertoleranter Systeme. Unter anderem erlauben die vorgestellten Modelle die formale Spezifikation der Hardwarearchitektur, eine präzise Definition der Softwarekomponenten und ihres zeitlichen Verhaltens, die eindeutige Festlegung von Fehlerannahmen, sowie die Auswahl geeigneter Fehlertoleranzmechanismen.

Unter Verwendung eines vorlagenbasierten Codegenerators wird der entsprechende Code erzeugt. Die Erweiterungsmöglichkeiten des Codegenerators bieten eine Lösung für die vorherrschende Heterogenität der verwendeten Ausführungsplattformen. Die Arbeit stellt eine generische Softwarearchitektur vor und diskutiert die Umsetzung bekannter Fehlertoleranzmechanismen. Zudem wird aufgezeigt, wie formale Methoden zur Verifikation integriert werden können.

Anhand von zwei aussagekräftigen Anwendungen werden schließlich die Vorteile des Ansatzes hervorgehoben. Die erste Anwendung zeigt, dass auch Systeme mit Regelungszeiten im Bereich von wenigen Millisekunden basierend auf Standardhardware unterstützt werden. Die zweite Anwendung demonstriert, wie durch eine Kombination mit existierenden Werkzeugen eines der wesentlichen Ziele im Bereich Software Engineering erreicht werden kann: die hundertprozentig modellbasierte Entwicklung - hier im Kontext von verteilten, fehlertoleranten Systemen.

Abstract

The design of fault-tolerant real-time systems is a complex task. The system must not only satisfy real-time requirements, but it must also deliver the specified functionality in the presence of both hardware and software faults. To achieve fault-tolerance, the system has to use redundancy. This redundancy is usually achieved by replicating hardware units and executing the application within a distributed system.

Model-based design tools promise to reduce the complexity of the design process by raising the abstraction level. However, most of the existing tools focus only on functional aspects. Code realizing non-functional requirements such as fault-tolerance mechanisms, communication, and scheduling is not targeted. However, this type of code makes up the majority of the code of a fault-tolerant real-time system.

This work presents a model-based development tool for the design of fault-tolerant real-time systems. The tool focuses on the code generation of non-functional requirements and therefore complements existing tools. The major contribution of this thesis is the presentation of adequate models that can be used to model fault-tolerant systems and generate the code automatically. These models comprise a formal description of the hardware architecture, the software components and their temporal behavior, the fault assumptions, and the selected fault-tolerance mechanisms.

Using a template-based code generator, the fault-tolerant real-time system is generated. The template-based code generator allows an easy expansion of the code generation functionality and therefore offers a solution to handle the heterogeneity of fault-tolerant systems. The thesis presents a generic architecture for fault-tolerant systems and discusses the realization of well-known fault-tolerance mechanisms in this context. Finally, the thesis outlines how formal methods can be integrated to prove the correctness of the generated code.

Two complementary applications are used to demonstrate the practicability of the approach. One application points out that control times in the range of a few milliseconds can be achieved using standard hardware. The second application demonstrates that by combining different tools, one major goal in software engineering can be achieved: the development of a complex and distributed embedded system in a complete model-based way.

Acknowledgements

First of all, I want to thank my supervisor, Professor Alois Knoll, for providing me the opportunity to prepare this thesis and for supporting my work with ideas, criticism, motivation, and guidance. I am very thankful to Professor Edward Lee for accepting to be my external reviewer.

Many thanks go to the entire group Embedded Systems and Robotics at the TU München for the valuable discussions and the pleasant atmosphere. In particular, I want to thank Simon Barner and Michael Geisinger for proof-reading the thesis, as well as Markus Rickert for answering all my questions regarding the layout.

I also thank my family for accepting my role as a lost son in the last months. Thanks to my parents Erika und Manfred for the wonderful education and all the opportunities they offered me. Many thanks go to my sister Sabine who helped me proof-reading this thesis.

Finally, very special thanks go to my girlfriend Katharina for supporting me, especially during the last months. She gave me the extra strength, motivation, and love necessary to get this thesis done.

Contents

List of Figures

CHAPTER 1

Introduction

Contents

1.1 Background and Motivation

Significant advances of semiconductor technology with a drastic increase of performance and a decline of the prizes accelerate the replacement of mechanical and hydraulic solutions by computer-based control systems. In general, these systems must satisfy two requirements: they must meet hard real-time constraints, but also tolerate both physical faults (in hardware) and design faults (in hardware or software). The reason for these requirements is the application area of such systems: the systems control physical processes. An incorrect or delayed result may lead to monetary loss, damage, or even an endangerment of life. The systems must therefore be fault-tolerant real-time systems.

Experiences, for example in the avionic industry with fly-by-wire, demonstrated the feasibility of replacing mechanical and hydraulic solutions by computer based control systems [Phi03]. However, the development of such fault-tolerant real-time software is much more time- and cost consuming (by a factor of 20-30 [Nil04]) than the development of standard information systems with a comparable complexity. This additional effort can be explained by the system's architecture. Redundancy must be applied to achieve fault-tolerance. The redundancy is usually achieved by executing replicated software units within a distributed system [Pow94]. Fault-tolerant systems are there-

fore typically distributed and very often based on heterogeneous hardware. Because the complexity of such systems is dramatically increasing, the traditional way to implement fault-tolerant systems from scratch is not feasible anymore [VSK05].

The main fault-tolerance mechanisms were already developed between the 1950's and the 1970's [ALR01, Pra96]. However, state of the art is the re-invention of the wheel. Due to the necessity to adapt the fault-tolerance mechanisms to application requirements, there are only a few approaches to reuse mechanisms to reach fault-tolerance. Most of these approaches are based on specialized, very expensive hardware, such as the Siemens Simatic S7 [HO05, HG05]. Software implemented fault-tolerance (SWIFT) [BSW+00, RCV+05] offers a more flexible approach. However, these approaches are often very restrictive in their application domain or in the tolerated faults. A typical restriction is, for instance, the assumption that all components behave fail-silent.

Component-based development [Szy02] offers a reduction of the development complexity by reusing components designed for fault-tolerant systems. Several research projects, such as GUARD [PABD+99] or DEAR-COTS [PVW04], use this approach. However, these approaches require the application developer to have in-depth knowledge on the concrete implementation of the components.

Computer aided software engineering (CASE) tools are one solution to address this problem. The topic of this thesis is the design of an adequate tool chain for fault-tolerant real-time systems. This goal is not new: already in 1988, Stankovic asked for a coherent treatment of correctness, timeliness, and fault tolerance in distributed computing [Sta88]. The goal of this thesis is that programming fault-tolerant systems using the presented tool chain becomes almost as easy as the development of classical non-fault-tolerant systems. Similar to an operating system, a fault-tolerant layer should be provided that delivers the necessary services to the application. This approach resembles the requirements stated by Stankovic:

> *The operating system must provide basic support for guaranteeing real-time constraints, supporting fault tolerance and distribution, and integrating time-constrained resource allocations and scheduling across a spectrum of resource types, including sensor processing, communications, CPU, memory, and other forms of I/O [Sta88].*

The approach should be based on the use of commercial-off-the-shelf (COTS) components, instead of relying on expensive, specialized hardware. Since a generic layer is not feasible due the heterogeneity of fault-tolerant systems, the idea is to generate the layer based on a high-level description of the application. Arora [AK98a] stated that the generation of components that add fault-tolerant properties to the system would reduce the complexity of the design process drastically. The approach is similar to the one of TinyOS. TinyOS is a highly adaptable middleware/operating system with adequate development tools for sensor networks. Analog to TinyOS, the tool should be used for the development of fault-tolerant real-time systems. Hence, the tool is named FTOS. The name may be misleading: code generated by FTOS should not necessarily replace an operating system. For general purpose processors with sufficient resources, the generated code is typically based on a real-time operating system. For more constrained hardware in contrast, the generated code may as well replace the operating

system. The main idea of this thesis and FTOS is that the fault-tolerance layer is generated tailored for the used hardware and the application. High-level descriptions of the system are used to extract the necessary information.

The model-based development paradigm [MM03] shares the same goals. For standard computer systems, it has become state of the art in software engineering. It helps on the one hand to cope with the increasing complexity of software systems by raising the level of abstraction and accelerates the development process by offering extensive code generation abilities. Therefore, the model-based paradigm is a suitable basis for the intended goal.

Different model-based tools are available for the development of embedded systems. However, these tools do not reflect the requirements posed by development processes of fault-tolerant real-time systems. The majority of the tools does not provide extensive code generation. One reason is the lack of precision in the semantics of the used models. The Unified Modeling Language (UML) [Obj07b] for example lacks the precision and rigor needed for extensive code generation [JSEB04]. Domain-specific tools like Matlab/Simulink [Bar05] or SCADE [DSMG04] offer code generation, but are limited to the generation of platform independent code like ANSI-C. Due to the limitation to platform independent code, these tools cover only the application functionality of embedded systems. In contrast, non-functional aspects such as fault-tolerance mechanisms, distribution, or temporal behavior are rarely covered and if so only for some specific hardware platforms or operating systems.

Nevertheless, exactly these non-functional aspects introduce most of the software complexity of fault-tolerant systems. A major reason, why these aspects are not covered by existing tools, is that the related code cannot be implemented without platform dependent operations or operating system calls. Therefore, the implementation depends on the concrete platform used. However, the deployed platforms, which is the combination of hardware components and an operating system, are highly heterogeneous. It is obvious that it is impossible to design a code generator that supports a priori all possible platforms. Nevertheless, this intrinsic heterogeneity of embedded systems is the primary challenge, as well as the ultimate justification for model-based design [SSBG03].

A solution to handle the heterogeneity is to design an expandable tool chain. A template-based code generator offers this expandability, as shown in [BKS05b, BKS06a]. It allows the addition of new templates to support further platforms, even by the application developer. Nevertheless, the expandability of the code generation functionality alone is not sufficient. Additionally, the modeling tools must be designed in an expandable way to allow the definition of additional fault-tolerance mechanisms. An expansion of the modeling language should of course not affect existing solutions. The required degree of expandability can be achieved by using a template-based meta code generator [VSK05]. This approach combines the advantages of both the model-based and component based approaches [TCC05].

The models used for code generation should be based on a high abstraction level and hide low-level implementation details. This is especially important as different stakeholders typically take part in the development of fault-tolerant systems. The models

should provide an unambiguous and precise specification of the system, to be a mean for the communication between different experts. Several requirements have to be satisfied by the modeling language to be suited for the domain of fault-tolerant real-time systems. These requirements are discussed in this thesis and a domain specific language [Lan66, Ram97], which satisfies these requirements, is presented.

Different fault assumptions are another reason why the reuse of components is so difficult. Fault assumptions describe the effect and probability of faults that have to be taken into account when designing a component. If a developer wants to reuse a component, he has to carefully analyze, if the fault assumptions of the own system are consistent with the assumptions used during the implementation of the component. This is a major reason, why the different component-based approaches require in-depth knowledge of the components. To automate the component selection as intended by this thesis, two requirements must be fulfilled. First, the fault assumptions must be specified formally and unambiguously. Second, formal methods must be applied to verify the consistency of the assumptions. The thesis points out how both of these requirements can be satisfied.

To demonstrate the practicability of the presented approach, two example applications are used. The first application is a balanced rod using switched solenoids. This application has stringent requirements on the control response times (in the range of few milliseconds) and is implemented on a triple-modular redundancy architecture. By this application, the efficiency of the generated code can be demonstrated. The second application is an elevator application consisting of a hot-standby system realizing the control logic and several field processors performing the I/O operations. This application demonstrates the possibility to use the tool to develop a complex distributed system. In addition, the thesis points out that by combining FTOS with other model-based tools to design the application functionality one major goal in software engineering can be achieved: the development of a complex and distributed embedded system in a complete model-based way.

The remainder of this chapter is composed as follows: the goals of this thesis are summarized in Section 1.2. Section 1.3 highlights the main contributions of this thesis. Concluding this chapter, an overview of this thesis is given in Section 1.4.

1.2 Goals of this Thesis

After motivating the topic of this thesis, the main goals are summarized in this section. As already stated in the title, the primary goal is the design and implementation of a **model-based tool with a sound formal basis** for the development of fault-tolerant real-time systems. Since there are a number of tools available for the development of the functionality of embedded systems, this thesis handles the **automatic generation of code for non-functional aspects**, in particular the generation of the fault-tolerance mechanisms.

Thereby, the goal of this thesis is to come up with a two-phased design methodology as introduced by Kopetz and Bauer [KB03] to complement existing tools. This methodology consists of an architecture design phase and a component implementation phase.

In the architecture design phase, the developer can specify the interactions among the distributed application components, the temporal behavior, and the deployed fault-tolerance mechanisms. The components realizing the application functionality have to be implemented by the application developer. This can be done by using existing development tools. The glue code (run-time system) to combine the components and to realize the non-functional aspects is generated based on the models of the architecture design phase.

By adding code for non-functional aspects automatically to the application, the overhead of implementing standard applications as fault-tolerant should be reduced to a minimum. The current overhead is predominantly a result of the different low level details for the realization of non-functional aspects. The major challenge in this context is the formulation of adequate models that allow both a **high-level description** hiding low-level implementation details and an **extensive code generation**.

Another important goal is to design an approach that is not restricted with respect to the tolerated faults or that requires the use of specialized hardware. The initial tool should support standard fault-tolerance mechanisms based on commercial-off-the-shelf hardware. Nevertheless, the tool should also allow **expansions** to support additional hardware platforms or new fault-tolerance mechanisms. To avoid a repeated re-implementation, it must be possible to add these new functionalities guaranteeing **reusability**.

Reusability of components used in fault-tolerant systems is complicated due to the potential of different fault assumptions. Components that rely on different assumptions may compromise the correctness of the system. Therefore, it is necessary to **employ formal methods** to prove the correctness of the generated code. Finally, the results should be validated by the use of **meaningful example applications**.

1.3 Main Contributions of this Thesis

The result of this thesis is a model-based tool for the development of fault-tolerant real-time systems called FTOS. The contributions fall into four different categories:

1. Presentation of a model-based development process including an adequate tool chain for the development of fault-tolerant real-time systems
2. Formulation of an adequate modeling language for the precise specification of these systems
3. Integration of formal methods for an automated verification
4. Exemplary realization of standard fault-tolerance mechanisms and other important system aspects, as well as the realization of meaningful demonstrator applications

In this thesis, a consistent, model-based development process including the tool FTOS for the development of the non-functional aspects of fault-tolerant real-time systems is presented. The different phases of the development and code generation are identified and realized within FTOS. The fact that experts of different domains take part in the de-

velopment is reflected both in the development process and in the presented modeling language.

This modeling language is based on domain-specific meta-models. The thesis identifies the requirements that have to be satisfied by these meta-models and presents appropriate meta-models. Sastry pointed out that one must address the design of the whole system to increase the productivity [SSBG03]. Relevant aspects for fault-tolerant systems are of physical (hardware), functional (software), environmental (faults), and conceptual (fault-tolerance mechanisms) nature. The thesis presents meta-models for each of these aspects and defines the relationship between them. The most important features of these meta-models are the use of an appropriate execution model in the software meta-model, a formal specification of the fault assumptions in the fault model, and a high-level description of the fault-tolerance mechanisms in the fault-tolerance meta-model.

As execution model, the concept of logical execution times that was proposed in the project Giotto [HHK03] is used. This concept is modified to match the requirements of fault-tolerant control systems. The most important changes are the introduction of sequential executions and global ports. A detailed description of these changes can be found in Section 4.4. A fault meta-model forces the user to specify the assumptions on the expected faults in an unambiguous way. This contrasts the current state of the art, where the assumptions are specified informally, typically by some textual description. The specification is based on application-independent fault effects.

Based on the thorough definition of the model´s execution semantics and the effect of faults, formal methods can be integrated to automate and verify the correct selection of fault-tolerance mechanisms. The concrete implementation of the fault-tolerance mechanisms can be specified in a formal way using the predefined fault effects. The resulting formal model reflects the behavior of the generated mechanism in the presence of faults. A mismatch between different fault assumptions can be detected automatically. Since the verification is not the primary focus in this work, a simplified prototype is implemented that integrates formal methods in FTOS and allows the verification of basic properties. In particular, a black box approach with respect to the application functionality was used. To get an overall valid proof, the behavior of the different components realizing the application´s functionality must be integrated as well. This work will be done in a separate thesis based on the presented results.

The code generator of FTOS is designed using the component-based paradigm. Templates provide solutions for different aspects of the system for a specific platform and can be selected and adapted by the code generator. This approach realizes on the one hand the required expandability and increases on the other hand the reusability. Standard fault-tolerance mechanisms are analyzed in this thesis and the building blocks are identified. Starting from the concept to describe the fault-tolerance mechanisms as a set of detectors and correctors as suggested by Arora et al. [AG93, AK98b], the thesis augments and concretizes this concept. Besides the implementation of major fault-tolerance mechanisms such as triple-modular redundancy, hot-standby, and rollback recovery [Pra96], the thesis also focuses on the realization of an appropriate scheduling scheme. The result is a scheduling algorithm that combines the advantages both

of Earliest-Deadline-First [LL73] and Cyclic Executive [BS88], namely robustness and fault-tolerance.

Two complementary applications are used to highlight the advantages of the presented approach to generate very efficient code and to cope with the inherent distribution of fault-tolerant real-time systems.

1.4 Structure of this Thesis

The thesis is composed of seven parts discussing the different aspects. After this introduction, the thesis continues with a definition of the relevant terms and a description of the different concepts in Chapter 2. Furthermore, the chapter introduces a formal basis that is used in the following to formalize the different concepts.

Chapter 3 presents the development process of the tool. The focus is on the techniques used for modeling and validating the models, as well as on the code generation and the verification.

Chapter 4 describes the used meta-models. It starts by identifying the requirements that need to be satisfied in the context of fault-tolerant real-time systems. Afterwards, the different meta-models of FTOS are explained. Solutions for all described requirements are presented. For illustration purpose, the concrete models of the two example applications are discussed.

In the following chapter 5, a generic software architecture for the generated run-time system is presented. The run-time system executes the application components implemented by the developer in accordance with the specified model and realizes the fault-tolerance mechanisms. The components of the run-time system are identified and their functionality is explained. The focal point lies on the realization of the fault-tolerance mechanism and the realization of an adequate scheduling. The benefits of the introduced approach are described at the end of this chapter by evaluating the outcomes of the demonstrator applications.

Finally, Chapter 6 discusses the verification of the generated run-time system. The main idea is to define a formal model that reflects the behavior of the generated components. Using a SMT (Satisfiability Modulo Theories) solver, different properties of the system can be proven automatically. The chapter points out the feasibility of this approach to prove the correct selection and implementation of fault-tolerance mechanisms with respect to the fault assumptions.

The thesis is concluded with a summary of the main results of this thesis. In addition, several starting points for future research are identified.

For interested readers, Appendix A provides an overview about the related work mentioned in this thesis.

CHAPTER 2

Background: Fault-Tolerant Computing

Contents

This chapter gives an introduction into the concepts related to fault-tolerance mechanisms and defines the most important terms used in this thesis. In addition, a formal model of systems, faults, and fault-tolerance mechanisms is presented. This model is used as the basis for the formal verification of the generated run-time system.

2.1 Definitions

This section starts with some definitions to explain the terms used in this thesis and to provide the reader with a basic understanding of the different concepts. The definitions in this section are mostly based on Avizienis et al. [ALRL04], Pradhan [Pra96], and Lee [LA90].

A system `s` can be interpreted as a set of interacting components `C`. Following [LA90], a component is any identifiable mechanism with a specific behavior at its interface. The interface is defined as the logical place between the component and its environment, which can consist of one or more components. The execution of a component `C` can be threatened by **faults**, **errors**, and **failures**. The easiest way to illustrate the different notion of the terms fault, error, and failure is by explaining the differences using the three-universe model [Pra96]. This model, an adaptation of the four-universe model introduced by Avizienis [Avi82], describes the different phases of the evolution from a fault to a failure.

The first universe is the **physical universe**, where faults occur.

Definition 1 *A **Fault** is a physical defect, imperfection, or flaw that occurs within some hardware or software component [Pra96].*

Faults can be **dormant** for a long time and not influence the execution of the component. When a fault is activated, the effects can be observed in the **informational universe**.

Definition 2 *An **Error** is the manifestation of a fault [Pra96].*

Errors can be detected by the component itself, if some rules are defined to evaluate the state of the component. However, these tests may not be able to identify the cause of the error, the fault. Initially, errors are only reflected in parts of the component's state. If the error is not detected early enough by the component, the error may cause a subsequent failure.

Definition 3 *A **Failure** of a component occurs when the component deviates from the specified behavior [LA90].*

Therefore, the third universe is the **external universe**, where the deviation from the expected behavior of a component can be observed. Consequently, a failure is the event that can be detected by interacting components. Thereby, a failure of a component can be a fault to its environment.

There are various reasons for faults. For instance, a fault can be a design fault, a physical fault, or an operational fault. While design faults are always active, physical faults are activated spontaneously with a certain probability. Typically, this probability is assumed to be constant and denoted by λ. Faults can also be classified according to their effect. The effect can either be in the value domain or in the time domain [PCD91]. Faults in the time domain are, for example, lost or delayed messages in a communication channel, but also additional messages. Faults in the value domain are e.g. erroneous results or bit flips in a message.

Fault-tolerance is the technique to guarantee that despite the presence of faults, a system provides the specified behavior to its outer boundaries [Lap85]. Fault-tolerance is always based on the effective deployment of redundancy, additional means that are not necessary to provide the specified behavior in the absence of faults. Different types of redundancy can be used ranging from software or data redundancy, to time and hardware redundancy.

In this context, it is important to note that a mere replication allows only the toleration of operational physical faults. Design diversity must be employed [AC77] to be able to tolerate design faults. This is of course a very cost-intensive approach. Using code generators, which support the generation of different versions of components realizing recurring functionality, helps to make this approach more feasible. Even if the correct type and level of redundancy is selected, it is important to guarantee that a fault does not affect all redundant components, but is rather contained.

Definition 4 *A **Fault Containment Unit (FCU)** [HTBSL78], often also called fault containment region, is the set of components that are affected by a specific fault.*

It is the task of the fault-tolerance mechanisms to prevent the fault effects (errors) from spreading to other components than the components of the FCU. In contrast to other approaches like [Kop06], FTOS allows the hierarchical composition of FCUs. This enables the developer to define detailed fault-tolerance mechanisms for each fault containment unit. One example is that potentially erroneous memory might be tolerated by replicating the variables, while other faults affecting the computational unit that contains the memory might be tolerated at the level of this computational unit.

The different possible concepts for fault-tolerance mechanisms are known since the 1950's due to the unreliability of the components at that time [ALR01]. In general, one can divide the applied fault-tolerance mechanisms into four groups:

Definition 5 ***Error detection*** *allows the detection and localization of errors.*

Detecting an error is the first step to fault-tolerance. After an error is detected, the component has to analyze the affected subcomponents and the error type. This is essential to perform error recovery.

Definition 6 ***Error recovery*** *transforms a system state that contains one or more errors into a state without detected errors [ALR01].*

There are different mechanisms to perform error recovery. The two most prominent types are **rollback** and **rollforward** recovery. Rollback is realized by restoring a previous state of the component [Pra96]. This state was saved in a **checkpoint** before the component detected the error. Rollback recovery is a rather simple approach; the only difficulties arise in designing and generating the checkpoints. Especially, if several components must be set back, the realization may become tricky. Rollforward recovery uses application knowledge to compute a new, correct state out of the erroneous state. Usually, this transformation implicates a reduced quality of service.

Regardless of the concrete error recovery mechanism, it is essential to ensure that the same fault is not activated again.

Definition 7 ***Error handling*** *prevents local faults from being activated again.*

To correctly perform the error handling, the first step is the localization of the error and the identification of the cause, the fault. Within the second step, the fault is isolated by excluding the affected component from further interactions with other components. The affected component might be replaced by spare components. Further possibilities are to use other components to deliver the functionality in addition to the already delivered functionality or to degrade the system (**graceful degradation**). The isolated component can then be repaired, typically by an external agent.

If a sufficient level of redundancy is employed in the system, explicit error detection is not required. Instead one can use error masking:

Definition 8 ***Error masking*** *guarantees that programs continually satisfy their intended specification, even in the presence of faults [AK98a].*

Typical examples for error masking are hot-redundant systems: several redundant units are executed in parallel. By comparing results, errors can be detected. If the master unit is affected by an error, another correct unit immediately takes over the master's task. The erroneous unit is excluded and can be repaired in the following. After a successful repair, it is necessary to reintegrate the repaired unit into the system to preserve the intended dependability:

Definition 9 ***Integration*** *allows a repaired component to resume with its intended behavior and interaction.*

For a successful integration, the **state synchronization** is essential: all participating units must agree on a new system state. The correct implementation of the state synchronization is a very important, but also complicated, step.

Which of the above presented types of fault-tolerance mechanisms are applied depends on the dependency goals and on the fault assumption. Altogether, six objectives of dependability can be defined: availability, reliability, safety, confidentiality, integrity, and maintainability. This thesis focuses on safety and reliability, as they are the main goals of most systems. However, the proposed concepts can also be augmented and exploited to reach the other goals.

Definition 10 ***Safety*** *is the probability* $\mathtt{S}(\mathtt{t})$ *that a system either performs its function correctly or discontinues its function in a manner that does not disrupt the operations of other systems or compromise the safety of any people associated with the system [Pra96].*

Having safety as a goal might lead to the fact that a system is rather brought into a safe mode, typically an emergency stop, instead of continuing to deliver the specified function when the correct behavior cannot be guaranteed. This is the main difference in comparison to reliability:

Definition 11 ***Reliability*** *of a system is a function of time* $\mathtt{R}(\mathtt{t})$ *defined as the conditional probability that the system performs correctly throughout the interval of time* $[\mathtt{t}_0, \mathtt{t}]$ *given that the system was performing correctly at time* $\mathtt{t}_0$*. [Pra96]*

If the fault rate λ of a system $\mathtt{s}$ is assumed to be constant, the reliability $\mathtt{R}_\mathtt{s}$ of $\mathtt{s}$ can be derived from $\mathtt{R}_\mathtt{s}(\mathtt{t}) = \mathtt{e}^{-\lambda\cdot(\mathtt{t}-\mathtt{t}_0)}$.

As indicated, the concepts safety and reliability are contrary in the sense that a focus on safety might decrease the reliability and vice versa. Using hardware redundancy is a good example. A duplexed system, meaning that two redundant units are executed in parallel, can be used to achieve reliability or safety. Reliability can be achieved by executing both units in parallel and performing self-tests. If an error is detected, the affected unit is switched off and the other unit takes over the control of the system. In contrast, both units would compare their result to achieve safety. If a deviation in the result is detected, the system switches into fail-safe mode. Of course, the fail-safe mode must be described in the specification as well. A fault-tolerant component always tries to provide the normal/intended behavior. However, it might not be possible to deliver

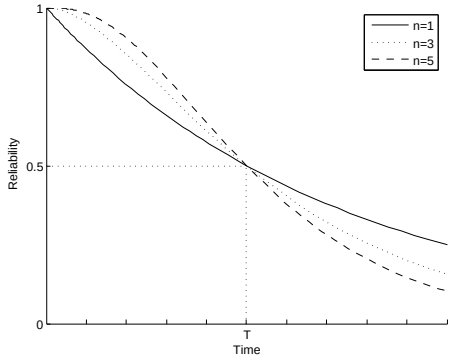


Figure 2.1: Reliability of Redundant Systems without Integration

this behavior in the presence of different faults. In such a case, the component should indicate an internal error to the environment and fail in a way that the environment is not harmed. This **exceptional behavior** [LA90, Lam81] must also be defined in the component's specification.

A classical example to combine reliability and safety is the use of a **triple-modular-redundancy (TMR)** architecture. Three redundant units are operated in parallel. With three correct units, the system performs a safe and reliable execution. If one unit fails, the two remaining units can continue with a safe, but not reliable execution. If again one unit fails or both units cannot agree on a consistent result, the system switches into fail-safe mode. Integration is an essential part in such systems. Without repair and integration, there exists a point in time, when the reliability of a TMR system falls under the reliability of a single-component system. This fact can be pointed out easily. For each of the components, the probability $\mathtt{R}$ that a component performs correctly within the period $[\mathtt{0}, \mathtt{t}]$ decreases with the increase of $\mathtt{t}$. Assuming a constant failure rate λ, the probability is $\mathtt{R} = \mathtt{e}^{-\lambda \mathtt{t}}$. Therefore, one can determine the point in time $\mathtt{T}$, when the reliability falls under $\mathtt{0.5}$. For simplicity reasons, the reliability of the redundant units is assumed to be equal. The reliability of the TMR system $\mathtt{R_3}$ can then be calculated by

$$R_3(t) = R_1(t)^3 + 3 \cdot R_1(t)^2 \cdot (1 - R_1(t)) \tag{2.1}$$

It is obvious that if $\mathtt{t} > \mathtt{T} \Rightarrow \mathtt{R_1(t)} < \mathtt{0.5} \Rightarrow \mathtt{R_1(t)} > \mathtt{R_3(t)}$. This fact can also be generalized for m-out-of-n-Systems, where at least $\mathtt{m}$ components of $\mathtt{n}$ redundant components must work correctly:

$$R_n(t) = \sum_{k=m}^{n} \left(\begin{array}{c} n \\ k \end{array} \right) R_1^k \cdot (1 - R_1)^{n-k} \tag{2.2}$$

The reliability of the different systems is also depicted in Figure 2.1. Of course, it may not even be possible to guarantee the exceptional behavior, if the number of faults exceeds a certain bound. How many faults of a specific type can be tolerated depends on the used mechanism, the type, and degree of redundancy used. However, it is not useful to select the type of redundancy without carefully analyzing the application. Different types of faults may never occur or are very improbable due to the concrete

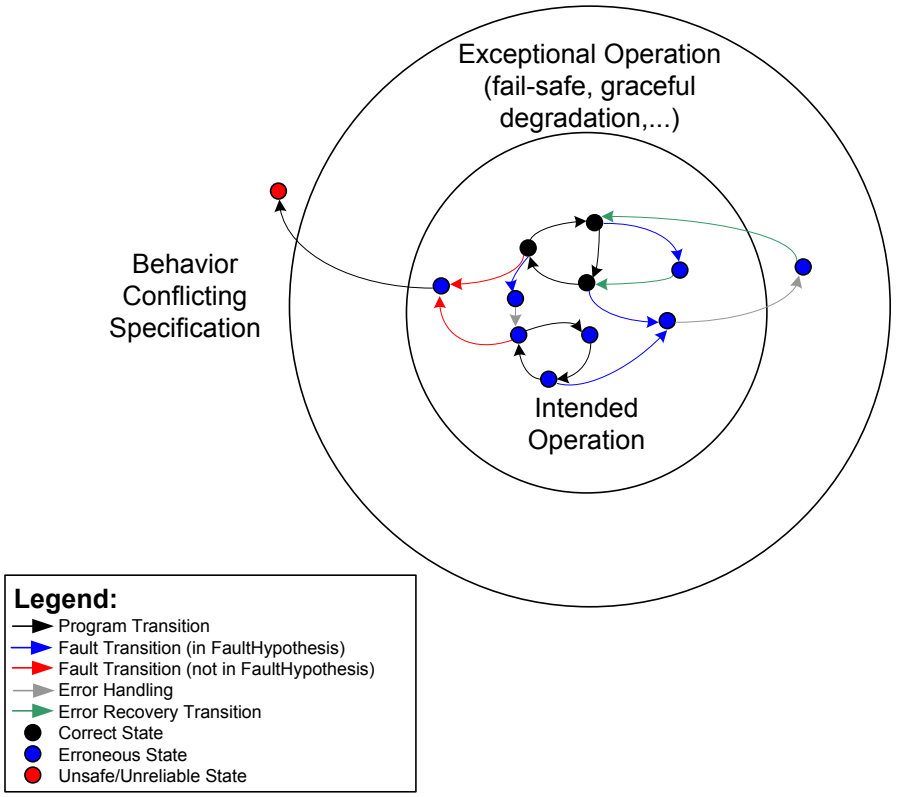


Figure 2.2: Fault-Tolerance Concepts

application scenario. A bit flip in a communication channel due to radiation effects may be a relevant issue for systems that are operating in outer space or for air crafts, but for a car the probability is negligible. The result of this analysis process is a **fault-hypothesis**.

Definition 12 *The **fault hypothesis** contains the assumptions about possible faults, their probability and effects to the components of the system.*

The assumed behavior of the affected components ranges from fail-stop [SS83], where the components simply stop providing services to other components, to Byzantine faults [LSP82, DHSZ03], where the components can exhibit arbitrary, even malicious behavior. The fault hypothesis $\mathtt{FH}$ can in principle be divided into two parts $\mathtt{FH}_1, \mathtt{FH}_2$. For less severe faults specified in $\mathtt{FH}_1$, the system is required to perform the expected behavior. This requirement comprehends both that the behavior $\mathtt{B_C}$ of a component $\mathtt{C}$ under the presence of faults specified in $\mathtt{FH}_1$ is functionally consistent and temporally acceptably consistent (since fault handling costs time) with the specification $\mathtt{S}$ [Web88].

$$FH_1 \Rightarrow B_C \equiv_{funct} S \wedge B_C \cong_{temp} S \tag{2.3}$$

For many real-time applications however, the components must behave temporally equivalent regardless of the presence of active faults. For such systems, only error

masking techniques are applicable. This is the reason, why the focal point of this thesis is on these techniques.

If the number or type of faults exceed the assumed numbers specified in FH_1, it is important to design the system in a way that it can detect these failures and switch into a fail-safe or emergency mode. The number and type of faults that should at least be detected are specified in FH_2. Figure 2.2 illustrates the discussed concepts. The system starts operating as described in the specification. If a fault is activated, the system's state is influenced by the effect of the fault. In the figure, it is assumed that the system is still performing as intended, but that the state of the system is then erroneous. However, it is also possible that the system fails immediately when a fault is activated. In the following, the system might detect the error, if the fault is covered by the fault hypothesis by performing tests. If the system detects an error, the system handles the error directly and repairs the system's state or switch into fail-safe mode. The system might be able to switch back from the fail-safe mode to the intended operation by applying error recovery actions or by external maintenance. It is however also possible that the error cannot be detected and that the system transitions immediately or subsequently to a state that violates the specification.

2.2 Formal Specification of the System

Based on the previous introduction of the most important terms and concepts of fault-tolerant computing, this section specifies a formal model of a fault-tolerant system including faults and mechanisms used to reach fault-tolerance. Based on the work of Arora et al.[AG93, AK98b], a formal framework is presented that can be used to specify the concepts introduced in this thesis formally. Major changes compared to the work of Arora et al. are to emphasize the component based nature of the discussed systems and to introduce concepts with respect to time.

2.2.1 System and State

A system $S = (V, \Pi)$ can be described by a finite set of variables V and a finite set of processes Π. The domain D_i is finite for each variable v_i. A state s of the system S is the valuation $(d_1, ...d_n)$ with $d_i \in D_i$ of the program variables in V. A transition is a function $tr : V_{in} \rightarrow V_{out}$ that transforms a state s into the resulting state s' by changing the values of the variables in the set $V_{out} \subseteq V$ based on the values of the variables in the set $V_{in} \subseteq V$.

2.2.2 Component

The system is build up by a set of components C. A set of variables $V_c \in V$ is associated with each component $c \in C$. $V_c = V_{c,internal} \cup V_{c,interface} \cup V_{c,environment}$ is composed by three distinct variable sets, the set of internal variables $V_{c,internal}$, the set of interface variables $V_{c,interface}$, and the set of environment variables $V_{c,environment}$. The internal variables can only be accessed and altered by Π_c. The interface variables are used for

component interaction and can be accessed by all interacting processes. The environment variables are the variables that are shared between the component and the environment of the complete system. This set can be again divided into the input variables $V_{c,input}$ that are read from the environment and the output variables that are written to the environment $V_{c,output}$.

Components can also be structured in a hierarchical way. A component $c \in C$ may consist of several subcomponents $c_1, ..., c_n \subset C$. The set of interface variables $V_{c,interface} \subseteq \bigcup_{1 \leq i \leq n} V_{c_i,interface}$ of c is a subset of the interface variables of its subcomponents $c_1 ... c_n$. The set of environment variables $V_{c,environment} = \bigcup_{1 \leq i \leq n} V_{c_i,environment}$ is the union set of all environment variables of the subcomponents.

2.2.3 Process

The functional behavior of a component $c \in C$ is reflected by a corresponding process Π_c. Let $V_{interface} = \{v_{c'} | v_{c'} \in V_{c',interface} \wedge c' \in C\}$ be the set of all interface variables. Π_c is specified as a finite set of operations of the form $guard \rightarrow transition$, where $guard : V_{guard} \rightarrow \mathbb{B}$ is a Boolean expression over a subset $V_{guard} \subseteq V_c \cup V_{interface}$ and $transition : V_{in} \rightarrow V_{out}$ is the appendant transition with $V_{in} \subseteq V_c \cup V_{interface}$ and $V_{out} \subseteq V_c \cup V_{interface}$. The thesis refers to the old value of a variable by v and to the new value by the primed variable v'.

Processes are expected to be deterministic, meaning that for each state s at most one $guard$ can evaluate $true$. This condition can of course be implemented easily by using one variable as a program counter and including this variable into the $guard$ expression.

However by allowing different processes to coexist simultaneously, non-determinism is introduced. There is no semantics, which process will perform its operation, if several processes have an enabled operation. While non-determinism should be excluded for the normal execution of the system, it is irreplaceable to model faults as Section 2.2.6 points out. The interplay between different processes describing the system can easily be implemented in a deterministic way by specifying adequate guards. To reach this goal, one might need auxiliary interface variables or use the value of time for time-triggered systems.

2.2.4 Time

The time is similar to the components reflected by one process Π_{Time} realizing the time progress and a variable v_{time} containing the current time. Π_{Time} reflects the logical time and cannot be affected by any faults. In contrast, the local time on the individual computational nodes is derived from components describing the behavior of the clocks used in the system, the related process, and its variables. The transitions can describe their temporal behavior by adapting the local time variable.

2.2.5 Specification and Predicates

Until now, the system is only considered in the absence of faults and without any fault-tolerance mechanisms. The first step to reach fault-tolerance is to translate the specification into a set of properties that must be valid for the application. While Arora et al. use computations, sequences of subsequent states, to express safety properties, the thesis uses state predicates P to express properties. A **state predicate** is a Boolean function over a set of variables $V_P \subset V$. Since the specification is defined implementation independent, the set of variables $V_P \subseteq \forall_{c \in C} \cup V_{c,environment}$ is a subset of all variables that can be observed by the environment of the system.

To express also temporal properties, it may be necessary to define auxiliary variables that record the progress of the environment variables over time. By explicitly stating these variables, a potentially unnecessary tracking of all variables can be avoided. In general, only very few variables are needed for the history state [AK98a]. In addition, liveness specifications can easily be expressed by state predicates using the time process Π_{time} instead of having to use complex liveness specifications, like the definition of Alpern and Schneider [AS85].

The transitive closure defines the fault-free system as depicted in Figure 2.2. It is defined as all states that can be reached beginning from some start states s_{start}. For all states within this transitive closure, the state predicates P describing the intended operation must be true.

2.2.6 Faults

The introduction of faults is straightforward and modeled as a component FH. The component is described as a set of variables $V_{c,FH}$ and processes Π_{FH} that perform actions in accordance to the fault hypothesis. The effects of a fault depend on the type of the affected component and are limited by the assumptions contained in the fault hypothesis. Due to the non-deterministic behavior of processes, the non-deterministic behavior of certain fault types can be perfectly matched. The propagation of an error depends in turn on the interaction between the different components and their implementation. Therefore, it is necessary to define the behavior of a component in the presence of faults. This can be done easily by changing the actions of Π_c for a specific component. Changes can be the introduction of new actions or the addition of conditions to a guard. Both, the additional elements and the new actions, can be based on the variables V_c and $V_{c,FH}$. A good example is fail-stop [AK98a]: an auxiliary variable up_c denoting the fault status of a component c can be introduced. For all actions of P_c, the guard is expanded with a condition $\neg up_c$ to restrict the execution only to the states, where the component is not affected by a fail-stop fault.

To allow a reuse of the components and their formal specification P_C, it is necessary to define the set of potential fault effects for the component class explicitly in advance. Section 4.5 points out the solutions of this thesis for this issue.

2.2.7 Fault-Tolerance Mechanism

Kulkarni and Arora [Kul99, AK98b, AK98a] pointed out that it is sufficient to use **Detectors** and **Correctors** to reach fault-tolerance. Detectors $\mathtt{d} : \mathtt{V_d} \subseteq \mathtt{V} \rightarrow \mathbb{B}$ are Boolean functions that monitor the variables of the system and can detect errors. Using the definition of predicates, a predicate D (detector) detects a predicate E (erroneous state), if the following conditions are satisfied for all possible sequences $\mathtt{s_0}, \mathtt{s_1}, ...$:

- **Safeness:** $\forall \mathtt{i} \geq 0 : \mathtt{D}(\mathtt{s_i}) \Rightarrow \mathtt{E}(\mathtt{s_i})$. This condition requires that, if the detector detects an erroneous state, the decision has to be correct. False positives are not accepted.
- **Eventual Detection:** $\forall \mathtt{i} \geq 0 : \mathtt{E}(\mathtt{s_i}) \Rightarrow \exists \mathtt{j} \geq \mathtt{i} : \mathtt{D}(\mathtt{s_j}) \vee \neg \mathtt{E}(\mathtt{s_j})$. This condition requires that a detector will eventually detect a permanent erroneous state.
- **Stability:** $\forall \mathtt{i} \geq 0 : \mathtt{D}(\mathtt{s_i}) \Rightarrow \mathtt{D}(\mathtt{s_{i+1}}) \vee \neg \mathtt{E}(\mathtt{s_{i+1}})$. The detector is also required to be stable: it should not signal the disappearance of an error, if the error is still present.

As the detection of erroneous states is highly-application dependent, the application developer has to specify most of the tests that should be applied. FTOS provides several standard tests that can be configured by the developer. The selection of appropriate tests and their configuration can be done at a very abstract level.

Correctors are implemented by actions `guard` $\rightarrow$ `transition` that transform an erroneous system into a system with correct state. The actions are triggered by a detected error. This notion is however very abstract, as a corrector may transform the system into an arbitrary state. While this approach is correct from a theoretical point of view, the system should of course continue the execution in a state that is most appropriate for the current erroneous state. The selection of this state can be forced by restricting the state changes through state predicates based on variables representing the state history. However, this approach complicates the specification of these state predicates.

In FTOS the notion of a corrector is concretized and augmented. Similar to Arora, the fault tolerance mechanisms are defined as actions `guard` $\rightarrow$ `transition`. However, different types of classes of mechanisms are introduced to support the goal of a generic approach. The main idea is to distinguish between operations that are executed online and operations that are executed offline. The concept of **error treatment** operations reflects the ideas of correctors. Error treatment operations describe the reactions of the system, when new errors are detected. The operation may be based on previously executed **proactive operations**, which are operations that are executed during the system execution to generate information redundancy, e.g. in the form of checkpoints. The introduction of proactive operations allows the separation of fault-tolerance concepts and application logic. Erroneous components are usually excluded from the system operation and can perform **error recovery** operations offline. After a successful completion of the recovery operations, the erroneous components can be integrated to guarantee the achievement of the reliability goals. The **integration** operations perform the state synchronization.

CHAPTER 3

Overview of the Approach

Contents

This chapter presents a holistic overview of the approach and gives an introduction into basic concepts of FTOS. FTOS is used to model the system and to generate the code related to non-functional aspects. The phases of a standard development process that are supported by FTOS are depicted in Figure 3.1. The designer is supported during system design by providing a specification language. FTOS validates the models and generates a tailored run-time system that provides the functionality for fault-tolerance mechanisms, communication within the distributed system, scheduling, and I/O operations. The components realizing the application functionality have to be implemented by the developer. A number of tools are available that can support the developer in this task.

This chapter is intended to provide an overview of FTOS and to identify the development steps. It starts by identifying the requirements that have to be satisfied by FTOS to achieve the intended goals. Based on these requirements, the basic concepts of FTOS are discussed and the different development steps are presented. At the end of the chapter, two applications are presented that are used to illustrate the introduced concepts.

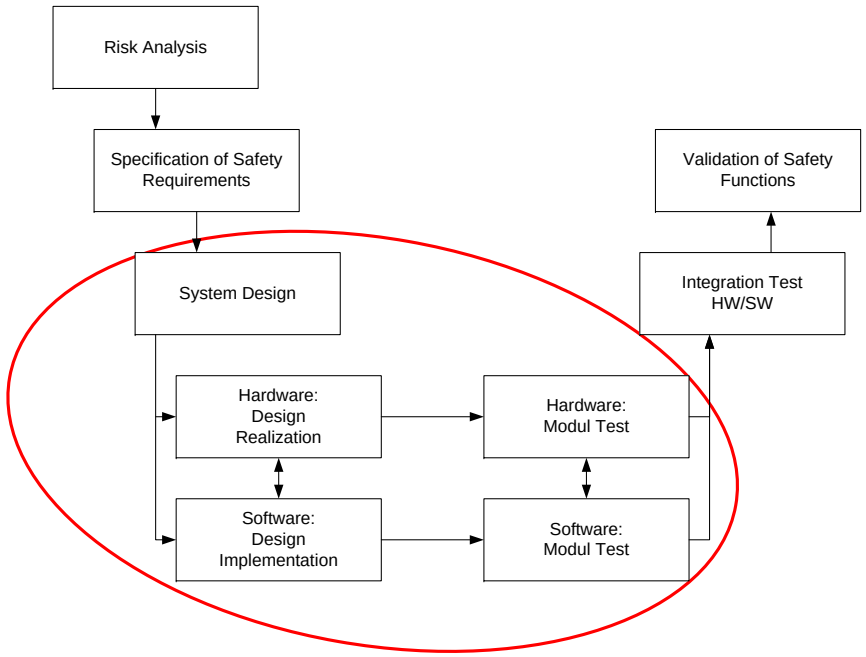


Figure 3.1: FTOS: Relevant Development Phases

3.1 Requirements on the Tool

This section identifies and discusses the main requirements on tools used for the model-based development of fault-tolerant real-time systems. As already stated in the introduction of this chapter, various tools are available for the development of embedded systems. Several of them are discussed in Appendix A. However, these tools focus mostly on the application functionality. Code realizing non-functional aspects in a fault-tolerant, non-monolithic real-time system has to be implemented manually. This code is necessary to realize fault-tolerance mechanisms, communication within the distributed system, I/O operations, scheduling, and process management. The main reason, why the generation of such code is not covered by existing tools, is the platform dependency of these mechanisms. The realization depends on the used operating system and hardware and cannot be implemented using platform independent programming languages like ANSI-C. Due to the great heterogeneity of used hardware and operating systems [SSBG03, Lee00], it is not possible to implement a code generator that supports a priori all possible combinations. This leads to the first requirement:

Requirement 1 *The code generator must be expandable (even for the application developer) to support additional platforms and arbitrary programming languages.*

Providing simple means to expand the code generation is the first step to get a useful generation tool for fault-tolerant real-time systems. However, an easy expansion of the code generation alone is not sufficient, since the initial modeling language can not cover

all possible mechanisms that one might want to realize utilizing the development tool. For instance, the tool supports the most important fault-tolerance mechanisms such as active and passive replication or rollback recovery. However, there are of course many other mechanisms that might be suitable as well. To support a new mechanism, the modeling language must be expanded to allow the specification of the required information. The same is true for the automatic generation of I/O operations. If in different projects the same device is used repeatedly, it might be reasonable to add generation functionality to the code generator to support this device. In case the class of the device is not supported in the current version, one might need to add a new device class with certain parameters in the modeling language. Therefore, the second requirement that must be satisfied by the tool is:

Requirement 2 *The modeling language must be easily expandable.*

It is important that expansions of the modeling language or the code generator must not affect the existing parts of the code generator. By allowing the expansion of meta-model and code generation functionality, the code generator can be adjusted to the requirements of the company or developer group using the tool. Such groups comprise typically different stakeholders, such as real-time system experts, hardware experts, safety manager and domain experts. This fact can be exploited similar to the approach in component-based approaches [Szy02, ABPG05]. The responsibility for different aspects of the code generation functionality can be assigned to dedicated experts. To support this approach, the third requirement must be satisfied:

Requirement 3 *The code generation functionality must be separated into modules to allow an independent implementation of solutions for different aspects.*

In addition, the multitude of different experts causes problems. Having different backgrounds and using different approaches, the interaction and communication between different experts plays an important role. Models are a natural connection factor. To ease the communication process, the used models must be simple, intuitive, and unambiguous. This requirement is even more strengthened by the fact that the models are used for extensive code generation. Many modeling languages such as UML [Obj07b] lack the precision and rigor needed for extensive code generation [JSEB04]. This leads to the next requirement:

Requirement 4 *The specification/modeling language must have explicit and unambiguous (execution) semantics.*

Since this requirement is very extensive, this issue is discussed separately in Chapter 4. Explicit and unambiguous models reduce the probability of design errors. However, the tool must support means to prove the correctness of the system. These proofs must be integrated in two dimensions: model validation and code verification. For the first issue, extensive tests must be employed:

Requirement 5 *Tests have to be integrated in the tool to check the validity of the model.*

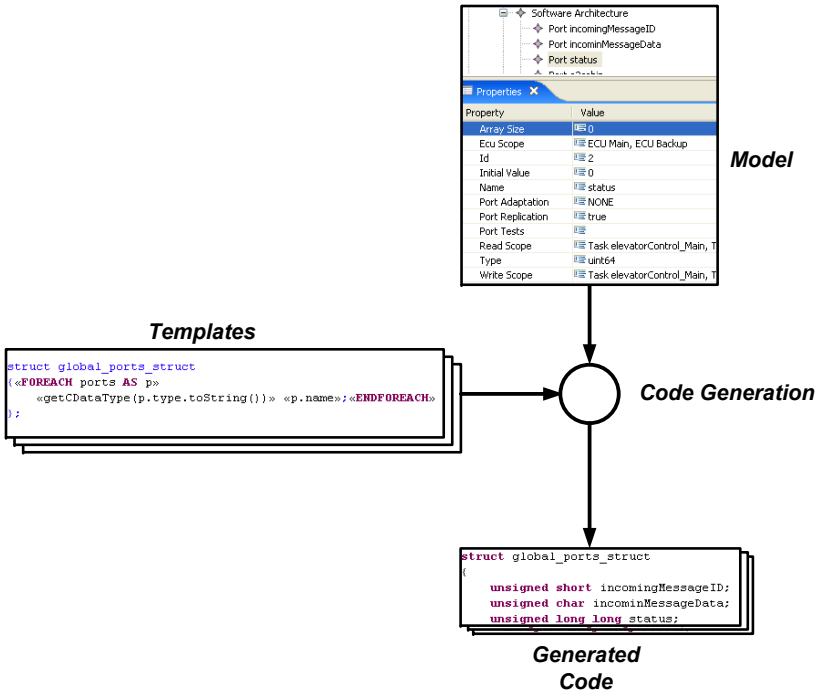


Figure 3.2: Principle of Template Based Code Generation

In addition, there must be high assurance in the generated code. In fault-tolerant system, especially the correct implementation of components taking into account the fault hypothesis is important. The same functionality might be implemented in different ways, if other assumptions on faults are used. Therefore, it is necessary to verify the generated code:

Requirement 6 *The tool should include formal methods to ensure a high assurance in the generated code.*

3.2 Template-Based Code Generation and Development Steps

This section enumerates the concepts used to satisfy the requirements stated in the previous section. Furthermore, the resulting tool is presented and the different components are explained.

The main requirement is an easy expandability of the code generator with respect to generation functionality and modeling language. The first requirement can be solved by leveraging template-based code generation [SSG91, BFVY96], as it was pointed out in the context of the predecessor of FTOS called Zerberus [BKS06b]. The concept of a template-based code generation is depicted in Figure 3.2. Instead of having one monolithic code generation kernel that encapsulates all the code generation functionality, template-based code generators consist of a code generation core and templates en-

capsulating the generation functionality. A template can realize a certain aspect of the fault-tolerant real-time system or can be used to combine further templates to finally form a complete run-time system. Thus, the input of the code generator consists not only of the model, but also of a set of templates. The task of the code generator is to analyze the model, select a suitable set of templates and to adapt these templates to application requirements.

The advantages of this approach are obvious: new templates can be added easily. These templates can realize new aspects of the system, e.g. support new hardware or a new fault-tolerance mechanism, but can also be used to generate code in a different target language. It is even possible to generate natural language to provide necessary documentation. Another advantage of this approach is that the complexity of the code generator can be reduced significantly. This is in particular very important when using the code generator for safety-critical system development. Very often, code generators are by far more complex than the generated programs. Thus, the certification of a code generator becomes usually too expensive. In addition, any changes of the code generator to expand the code generation functionality lead to the necessity of a new certification of the whole code generator. In contrast, template-based code generation solves this problem. The complexity of the code generation core can be reduced to a minimum. Furthermore, the addition of new templates only leads to a new evaluation of the affected templates, while other templates can be used without a repeated certification.

In addition, the concept of templates fulfills inherently the request for modularity. System engineers can define a generic software architecture and specify the interfaces between the different components. Experts can use their expertise to realize templates solving specific aspects of the complete system. Safety engineers can identify the important components of the system and add appropriate mechanisms to guarantee safety and reliability.

However, the concept of templates does not address the expandability concerning the modeling language. A solution is the use of a meta-code generation framework. Several of these frameworks are available such as openArchitectureWare [EVHK], AndroMDA[1] or MetaEdit[2]. These code generators allow the definition of modeling languages in the form of user-defined meta-models. The concept of these frameworks is depicted in Figure 3.3. Based on a meta-modeling language, the developer of the code generator can define a meta-model. In the example of the figure, a meta-model for finite state machines is described. The meta-modeling language is typically based on the class diagram notation and allows the definition of classes, references, attributes, and data types. Based on the meta-model, the application developer can define a concrete model. The support of object oriented concepts such as inheritance [GR83] and polymorphism [CW85] is a key factor to simplify the expansion of the modeling language / meta-model. By introducing new sub classes e.g. for a specific device class, the modeling language can be expanded straightforward. In addition, the concept of polymorphism allows this expansion by specifying a code generation function for that specific sub class. Other code generation functions can be left unchanged.

[1] http://www.andromda.org/
[2] http://www.metacase.com/

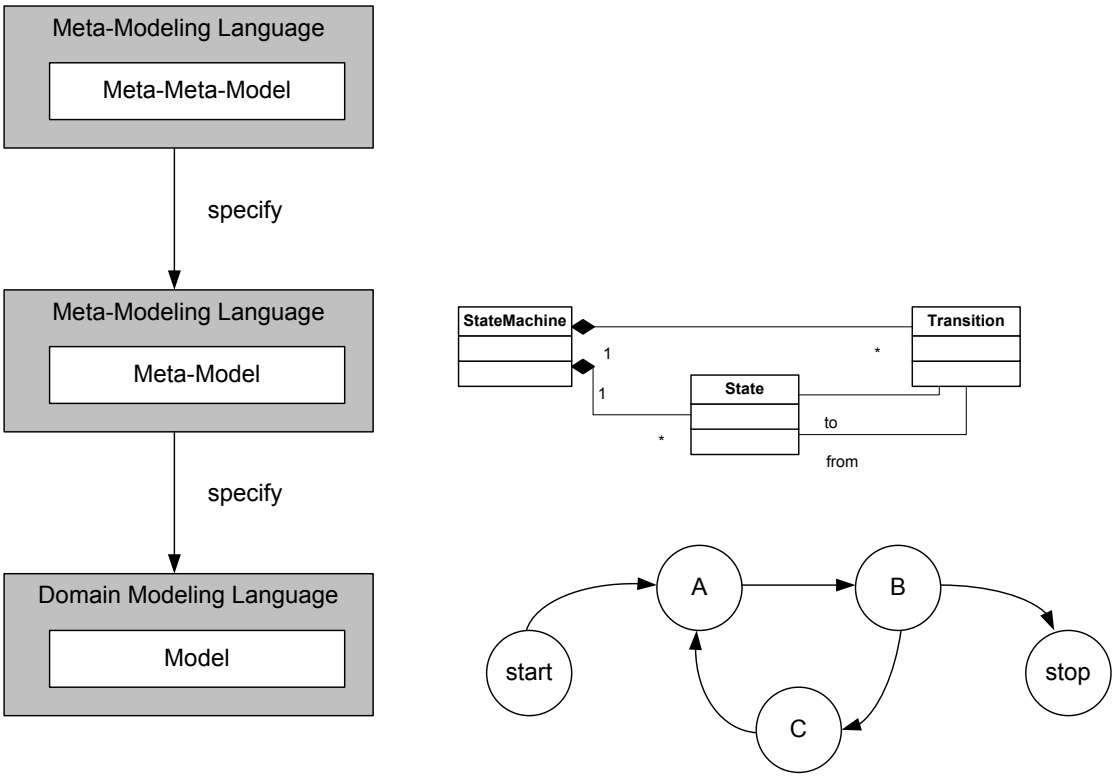


Figure 3.3: Meta-Modeling Framework Approach [Ins08]

Instead of augmenting the code generator Zerberus to support meta-modeling, FTOS is based on openArchitectureWare[3] (oAW) [VSK05]. The meta-models used in FTOS are described in Chapter 4.

The final two requirements are satisfied by incorporating model validation rules and formal verification. The validation rules are directly included in the code generation process. The verification is more complex. Since FTOS does not focus on a specific target language like Whalen et al. [WH99b], it is not possible to formalize the translation between modeling and target language. Furthermore, the verification must also be suited to support the expandability of the code generator. This dilemma is solved by specifying the formal behavior of a template. Based on this formal description, the formal model of the developed system can be generated in parallel to the original code generation process. Important properties can be verified by integrating formal verification tools. A detailed description of this approach can be found in Chapter 6.

The complete tool chain is depicted in Figure 3.4. Based on meta-models realizing the domain specific language, the developer team can specify the concrete system models. FTOS uses four meta-models to describe the different aspects of fault-tolerant systems. The modeling tool incorporated within oAW allows the specification of the models using graphical notations. Therefore, syntactical errors are excluded by design. Nevertheless, it is necessary to check the semantic correctness of the models. This model valida-

[3] http://www.openarchitectureware.org/

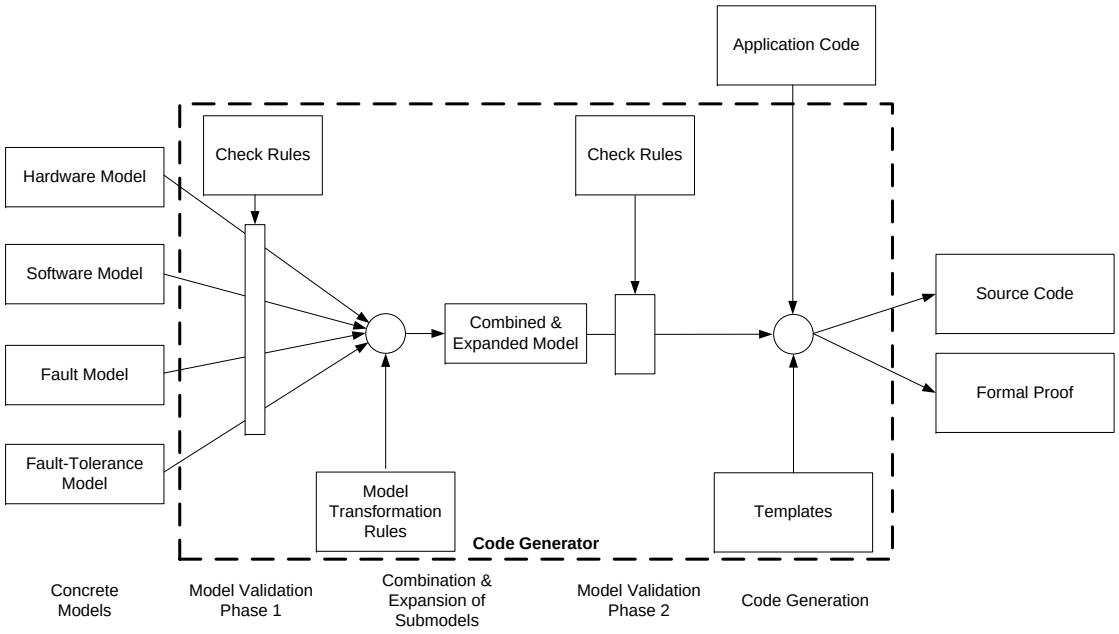


Figure 3.4: Code Generation Process

tion is realized in the next step. The validated models are then combined to one model. In addition to the mere combination, supplementary information is computed in this step to simplify the code generation. Some validation rules test the interaction between different models. These rules are checked after the model transformation. Finally, the validated, expanded, and combined model is used for code generation. The code generator selects appropriate templates to solve application aspects and adopts these templates to application requirements. The result is a tailored run-time system including mechanisms for scheduling, inter-process communication, fault-tolerance mechanisms, and synchronization. Furthermore, user implemented code that realizes the functionality for the application, such as a controller function, is embedded into the generated code. In parallel to this code generation, a formal model of the system is generated that can be used for formal verification. The individual steps are explained in more detail in the following section.

3.3 Code Generation Process

3.3.1 Modeling

The modeling activities in oAW are based on the Eclipse Modeling Framework (EMF) [BSM+03]. This modeling framework is used both for the definition of the meta-models and for the definition of the concrete models. The previous section already indicated that it is useful to split up the models into sub-models to describe the different aspects separately. This technique increases the simplicity of the models and the separation

of concerns. A detailed description of the used meta-models and their relation can be found in Section 4.

3.3.2 Validation

Semantic design errors have to be detected in early design phases. Therefore, several tests are formulated to check the validity of the specified models. The specification of these tests is supported in oAW by offering the validation language **CHECK**, an equivalent to the object constraint language (OCL) [Obj03] available for UML. Tests in CHECK are specified as formulas in First-Order Logic. One example for a test is depicted in Figure 3.5. This test checks whether each communication point (port) is read by at least one software component (actor). Other examples are tests to check the reachability of all application modes or to ensure the absence of constructs in the model that could introduce non-determinism.

3.3.3 Model-To-Model Transformation

The next step after the model validation is the combination of the different models. The resulting model is used for the code generation. To simplify the code generation, further information, which is already implicitly contained in the models, can be computed and added explicitly. One classical example is the handling of references: if the model contains an unidirectional reference, it might be useful to also add a reverse reference. This approach decreases on the one hand the error-proneness of the initial model, since directed references are in contrast to bidirectional references much easier to maintain for the developer. On the other hand, the code generator benefits from the bidirectional references. Model-to-Model (M2M)-Transformation is supported in oAW by offering the functional programming language **EXTEND**. In Figure 3.5, the result of the M2M transformation is depicted for a communication point. The M2M transformation computes the number of software components using the communication point and the number of relevant electronic control units. Further examples for the M2M transformation are discussed during the presentation of the used models in Chapter 4.

3.3.4 Code Generation

The code generation is based on templates, as described before. Templates represent the actual code generation ability and can be added easily. A template can be used to solve a certain aspect or to combine the results of different templates to form a run-time system. Most templates are platform dependent in the sense that they offer a solution only for a certain combination of hardware, operating system, and programming language. Therefore, also the correct selection of adequate templates is necessary.

oAW uses for the implementation of templates the `XPand` language. This language is very simplistic. It offers the statements `DEFINE` to declare a new code generation function and `EXPAND` to call other generation functions during the code generation. An important feature of oAW is the support of polymorphism to guarantee an easy expansion

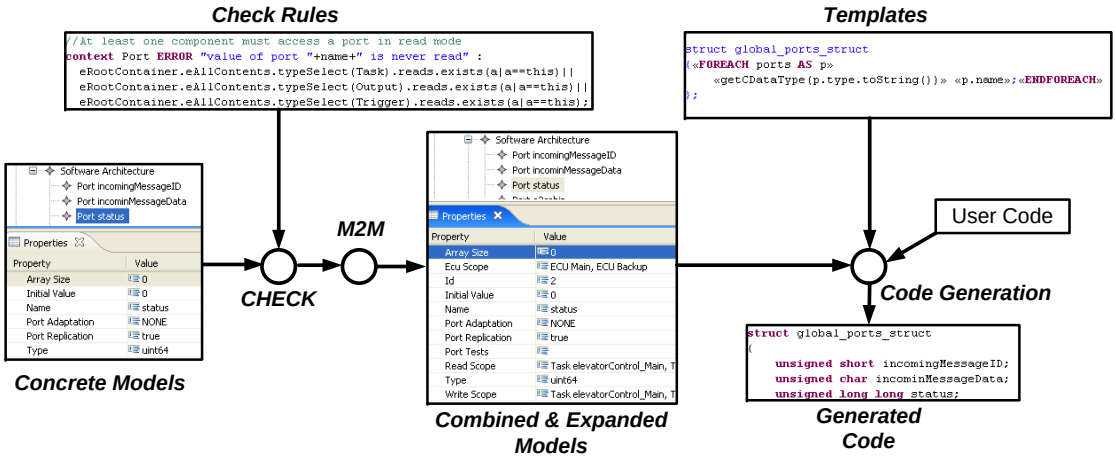


Figure 3.5: Code Generation - A Concrete Example

of the code generation ability. To specify the control flow of the code generation, the commands `FOR`/`FOREACH` and `IF`/`ELSE` can be used. The `FOREACH` statement is used to generate code for each object of a certain type that is declared within the model. Finally, the commands `FILE` and `ENDFILE` allow the management of the generated files.

The code generation technique itself is simple: the adaptation of the templates to the model is performed using a technique similar to preprocessor macros. Text sequences between the different XPand commands are directly copied to the generated files and variables allow the access to objects and their attributes. A description of the code generation technique can be found in [BKS06a]. Figure 3.5 shows one example for a template that is used to generate code for the declared communication points.

3.3.5 Verification

Formal verification is in particular of great importance for fault-tolerant systems to verify the correct implementation of mechanisms with respect to the fault hypothesis. Two components realizing the same functionality may be implemented in different ways, if the fault assumptions differ. However, the fault assumption of the system that should be developed does not necessarily match the fault assumption used for the implementation of the available templates. It is therefore necessary to assure the interoperability of different components for a specific application context. The main idea to solve this issue is to use a formal description of the components behavior. As presented in Chapter 4, the meta-model provides means to describe the behavior of components in the presence of faults. Because the formal description must be specified by template developers that have typically no expertise in formal verification, it is necessary to limit the required knowledge. This goal is achieved by using BoogiePL [DL05] for the specification. BoogiePL is actually an intermediate language for program analysis and program verification and resembles imperative programming languages. Developers have to learn very few concepts in order to be able to implement a formal specification. Based on this specification, the code generation tool generates a formal model of the complete

Figure 3.6: Demonstrator 1 - Balanced Rod

system. This formal model can be used to verify certain properties of the system by using a SMT (Satisfiability Modulo Theories) solver. More details on this approach are given in Chapter 6.

3.3.6 Code Generation Result

Usually, the generated files contain source code for an arbitrary programming language. But since oAW is not restricted to one specific output language, it is also possible to generate documents in natural language. This can be useful, if documents for certification issues or user-manuals are required. Currently, FTOS provides templates for the generation of executable run-time systems for two different platforms. These systems include code for the timely-correct execution of the application, for process management and scheduling, as well as communication (interprocess, interprocessor) functionality. In addition, the selected fault-tolerance mechanisms are realized by the run-time system. The actual code realizing the application functionality, like control functions, is not covered by the tool and has to be implemented by the developer. An overview of the generated code can be found in Chapter 5. A concrete example for the whole process is depicted in Figure 3.5.

3.4 Demonstrators

Two demonstrators are discussed in this thesis to point out the potential of the presented approach. The first demonstrator is a balanced rod using switched solenoids. The second demonstrator is an elevator control.

Balanced Rod

The balanced rod is depicted in Figure 3.6. The control application is implemented on a triple-modular redundancy (TMR) architecture guaranteeing safety and reliability of the system. The goal of this setup is to demonstrate the ability of the approach to generate efficient code. For a successful control of the application, the control response times

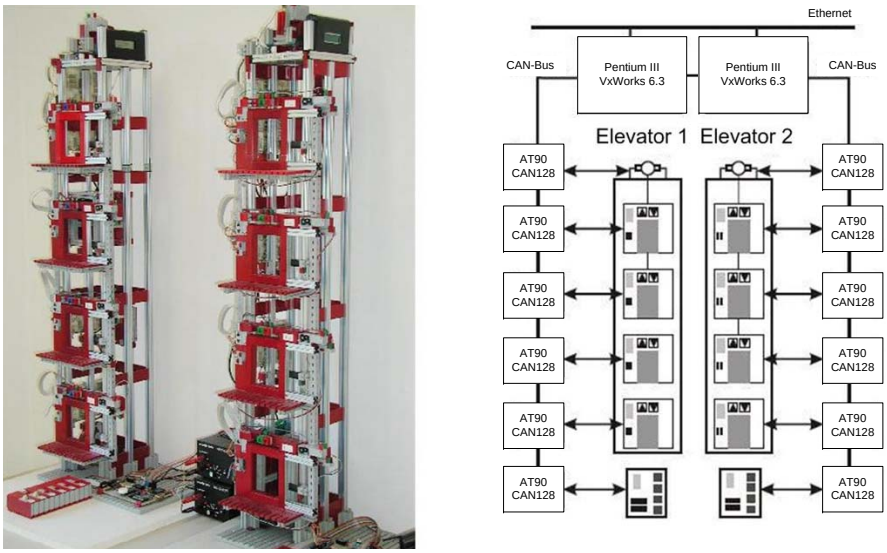


Figure 3.7: Demonstrator 2 - Elevator

must be in the range of a few milliseconds. Within this period, the sensor inputs must be processed, the control response calculated and voted within the TMR system, and finally the result must be output to the environment. The exclusion of erroneous nodes, the selection of a master, the temporal synchronization, and the integration of previously excluded nodes must also be realized within each time slot. The demonstrator results show the feasibility of FTOS for this application area.

Elevator Control

The setup of the elevator control demonstrator is depicted in Figure 3.7. The system consists of two control nodes executing the elevator control and several microcontroller (in the following called field controllers) realizing the interaction with the sensors and actuators. All computation nodes are connected via a Controller Area Network (CAN). This application is intended to point out the possibility to use FTOS for the development of complex, heterogeneous, and distributed systems. Furthermore, Furthermore, the example shows that FTOS can be combined with other model-based tools to achieve an integrated model-based approach.

CHAPTER 4

Adequate Meta-Models

Contents

This chapter presents the meta-models used in FTOS. The formulation of adequate meta-models is the key factor in designing a domain specific code generator. Therefore, the chapter starts by gathering the requirements. The general concepts and an overview of the used meta-models are discussed in Section 4.2. In the subsequent sections, the different meta-models are explained in detail. At the end of this chapter, Section 4.7 summarizes the concepts.

4.1 Requirements on the Meta-Model

Several requirements have to be fulfilled by the meta-model to achieve the identified goals. Bondavalli [BFLS01] pointed out that most design practices in the domain of fault-tolerant computing suffer from partial approaches. Instead of restricting the approach on some aspects of the whole system, it is important to come up with a system engineering approach [SSBG03]. This approach must comprise various aspects: information about the hardware components must be available to generate adequate code. The topology of the network must be known to realize mechanisms related to distributed computing. The software components, their interaction and the tempo-

ral behavior is required to generate a tailored run-time system including scheduling. The developer must have the possibility to specify the fault-tolerance mechanisms that should be added. The concrete implementation depends on information about the fault assumptions. Together, this forms the first requirement:

Requirement 1 *The meta-model must allow the specification of different aspects: the hardware architecture, a description of the software components, the fault hypothesis, and the fault-tolerance mechanism.*

Typically, these aspects are specified by different experts involved in the project: hardware specialists, experts in the application domain, system engineers and safety experts. It is necessary that all experts involved in the modeling task are able to understand the different concepts. Therefore, a high abstraction level is recommended. However, the possibility to raise the level of abstraction is limited by the necessity to automate the code generation: the model must be *generative* [KSLB03].

Requirement 2 *The right level of abstraction must be chosen to guarantee on the one hand the comprehensibility, but to preserve on the other hand the possibility to generate code. In particular, the meta-model must have clear and unambiguous semantics.*

While the requirement of unambiguous semantics is common for all models designed for code generation, there are also several requirements arising from the intended domain of fault-tolerant real-time systems. The timing behavior plays an important role in such systems [Sta88, Lee05]. Due to the fact that fault-tolerant systems typically consist of concurrent processes, the execution semantics of the models must be clearly specified to avoid race conditions.

Requirement 3 *The used model must have clear execution semantics and must contain explicit information about the temporal behavior.*

With respect to the generation of fault-tolerance mechanisms, additional requirements have to be satisfied by the model. Fischer, for example, showed the impossibility of reaching consensus in asynchronous systems [FLP85]. Consensus protocols however are the foundation of major distributed fault-tolerance mechanisms. It is also not sufficient to use algorithms, such as in the area of the Paxos problem [Lam98], that reach a consensus eventually. This would contradict the requirements of real-time systems. An adequate model of computation has to simplify the implementation of fault-tolerant systems significantly [WLG+78].

Requirement 4 *The model of computation must be suited for the realization of distributed fault-tolerance mechanisms.*

As already mentioned, most of the fault-tolerance mechanisms are based on some form of consensus. To automate consensus algorithms, it is necessary to extract information about the state of one unit. The same requirement arises in the context of state synchronization during integration. It is therefore necessary to identify the components of the model that reflect the state of the system.

Requirement 5 *The state and the functional behavior of the system must be separated within the model to allow state synchronization and voting.*

By separating the state from the behavior, it is possible to automatically identify the state values that must be compared during voting/consensus or state synchronization. In order to perform a successful voting, it is also necessary to achieve replica determinism [Pol94, PBWB00]. *A system is said to be deterministic, if for each possible state, and each set of inputs, a unique set of outputs and next state of the system can be determined [Lap93].* It is already difficult to achieve determinism in the context of embedded system, but replica determinism is even harder. Replica determinism describes the fact that two replicas behave the same in the absence of errors. This becomes a hard problem, if the replicas are executed in a distributed system. Non-determinism is introduced in such systems for example by small clock synchronization errors or by receiving events in a different order. Similarly, design diversity among replicas [Vog88] may lead to non-determinism. However, design diversity must be supported to allow the toleration of design errors. Therefore, one can state the following requirement:

Requirement 6 *The model must offer replica determinism without restricting design diversity among replicas.*

The previous requirements target the system during correct execution. To support the correct implementation of the mechanisms, it is however important to have knowledge about the faults that need to be tolerated by the system. The tool must therefore force the developer to state the fault assumptions. Unfortunately, the state of the art to record these assumptions is based on textual, very often ambiguous documents. In contrast, precise and unambiguous fault assumptions are required to support a correct code generation and verification.

Requirement 7 *The model must support a precise and unambiguous specification of the fault assumptions.*

Based on this knowledge, it must be possible to integrate formal methods to show the correctness of the models and the generated code. Bondavalli [BFLS01] pointed out that the formal verification of non-functional aspects is a key issue in the design of fault-tolerant systems. The models must be *descriptive* [KSLB03]:

Requirement 8 *The model must allow formal analysis, verification, and validation.*

4.2 Overview: Used Meta-Models

After stating the requirements, this section provides an overview on the used models. Bondavalli pointed out that for the design of fault-tolerant systems a system view is required [BFLS01]. Since the different aspects are mostly orthogonal to each other, it is however not useful to create one global meta-model. Instead the thesis proposes to

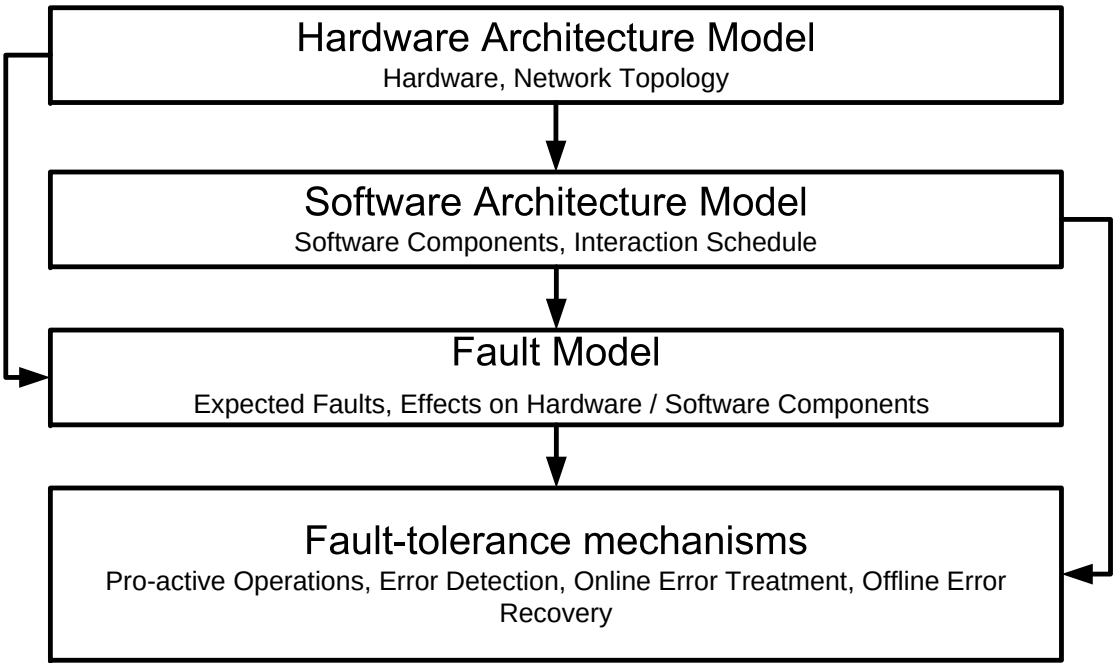


Figure 4.1: Model Dependencies

decompose the meta-model into different sub-models. This approach also reflects the fact that different stakeholders are involved in the development. Each expert group is then responsible for their sub-model. By defining the interfaces of the different models in an unambiguous way the system approach becomes feasible. In the area of fault-tolerant computing, four different aspects / sub-models can be identified:

- A **Hardware Architecture** model specifies the relevant information about the hardware components and the network topology.
- A **Software Model** describing the software components, their interaction, and the temporal behavior.
- A **Fault Hypothesis** contains all assumptions about the expected faults, their effects, and the expected fault configurations.
- A **Fault-Tolerance Mechanism** model specifies the mechanisms that should be employed to tolerate the expected faults and to guarantee the safe and/or reliable execution of the system.

For all these aspects, FTOS provides according meta-models. The meta-model design was directed by the two goals simplicity of the models and effectiveness with respect to code generation. These goals were reached by focusing on the specific characteristics of fault-tolerant real-time systems and by the formulation of a domain-specific meta-model. The potential conflict between simplicity and effectiveness is solved by using the M2M transformation.

The relation between the different sub-models is depicted in Figure 4.1. The approach differs from the standard model-driven architecture (MDA) as suggested by the OMG

[MM03] in the sense that FTOS starts with the hardware architecture. In contrast, in MDA the developer starts with a platform independent model. This model is then iteratively refined and augmented with platform specific details. Nevertheless, this strategy is not appropriate for modeling fault-tolerant systems. The first phase in the development process of fault-tolerant system is the analysis phase. Within this phase, the safety/reliability requirements are analyzed. These requirements determine the used fault-tolerance mechanisms and the hardware architecture. In this respect, the hardware architecture is already known, when the developers start the modeling phase in FTOS. Nevertheless, FTOS tries to preserve the benefits of a platform independent software design by restricting the platform dependencies to a minimum. At functional level, the developer has to make no assumptions at all about the platform. The only platform dependencies are the concrete software-hardware mapping and other optional information such as worst-case execution times. This information is however only required for the concrete implementation. Provided that the enough hardware resources are available to execute the system, the system behaves the same independent of the concrete platform in the absence of faults.

Based on the hardware and software, the developer can specify the fault hypothesis. This model contains assumptions about expected faults and their impact on hardware or software components. The approach forces the user to specify all assumptions in an unambiguous way. This is especially important, since the implementation of fault-tolerance mechanisms depends on the concrete fault assumptions [Kop06].

Finally, the developer can specify the fault-tolerance mechanisms. For this meta-model, FTOS augments the work of Arora and Kulkarni [AK98b, Kul99], which points out that **Detectors** and **Correctors** are enough to provide fault-tolerance. However, this model is not sufficient to provide an automatic generation of fault-tolerance mechanisms, because relying only on detectors and correctors would lead to a mixing between application logic and fault-tolerance mechanisms. Besides detectors and correctors, FTOS introduces therefore concepts for pro-active operations, off-line error recovery, and integration.

Preliminary Definitions

The following chapters give some details on the used meta-models. The meta-models define different classes that can be instantiated by the user in a concrete model. For each class, `ClassName` refers to the set of specified instances in the model. The abstract syntax defines the attributes and references of a specific instance. Most classes have a unique identifier (name) as attribute. In this case, the thesis refers to the instance simply by the identifier. An attribute is accessed in object oriented style: `name.attr` denotes the value of the attribute `attr` of the instance `name`.

The cardinality of an attribute or reference is indicated by the name. Each attribute / reference starting with a capital letter refers to a set that can consist of more than one element. All sets in FTOS are totally ordered. The relation $\texttt{contains} \subseteq \texttt{Element}^* \times \texttt{Element}$ determines whether an element is contained in a set. The function $\texttt{pos} : \texttt{Element}^* \times \texttt{Element} \rightarrow \mathbb{N}$ calculates the position of the element in the ordered set. Sets are defined recursively and each set is again an element. The

contains relation is similarly defined recursively. A set s contains an element e, if e is a direct element or is contained in one of the direct elements of s. If the element is contained recursively in a set, the function pos refers to the position of the direct element in the ordered set that contains the element recursively. To simplify the access on elements of the set, the operator get : $\texttt{Element}^* \times \mathbb{N} \rightarrow \texttt{Element}$ is available. $\texttt{get}(\texttt{s}, \texttt{n})$ returns the $(\texttt{n} + 1)$th element of the ordered set s.

4.3 Hardware Model

The motivation to specify the hardware architecture is two-fold. First, it is used similar to the concept of targets in Ptolemy Classic[BHLM94] to encapsulate knowledge about the specific platform. All information required for the proper selection and adoption of adequate software components, as well as for the realization of distributed mechanisms is contained in the model. This includes information about the computational units, the available devices, and information about the network topology, as well as used network protocols. Second, the information in the hardware model is also used for a formal verification of the whole system. The model defines the components that may in general be affected by faults. The information, which components are potentially affected by a fault, is contained in the fault meta-model.

A number of other modeling languages are available that cover hardware aspects such as the structure diagrams in the System Modeling Language (SysML) [Obj07a] or the Architecture Analysis & Design Language (AADL) [SAE04, FGHL04]. Most of the information contained in these models is not intended for the implementation of the software, which is the primary goal in FTOS. Therefore, FTOS proposes a simplified hardware model that is easy to comprehend and that contains only the necessary information.

4.3.1 Main Concepts

Figure 4.2 shows the structure of the hardware meta-model. At the highest level of abstraction, FTOS allows the definition of two objects: the computational units, in the following called electronic control units (ECUs), and networks.

Electronic Control Unit (ECU): The computational units are described by the class ECU. The abstract syntax for an ECU instance is $(\texttt{e}, \texttt{os}, \texttt{language}, \texttt{CPUs}, \texttt{Devices}, \texttt{clock}) \in \texttt{ECU}$, where e is the unique name of the unit, os defines the operating system running on the unit (if any), language refers to the programming language used for the run-time system and the additional references specify the included CPUs ($\texttt{CPUs} \subseteq \texttt{CPU}$), devices ($\texttt{Devices} \subseteq \texttt{Device}$) and the clock $\texttt{clock} \in \texttt{Clock}$. The information is used to generate a tailored run-time system for each such unit. The information on the used operating system and programming language are the initial criteria for the selection of an adequate run-time system framework.

Devices: One goal of FTOS is to automate the generation of the I/O functionality. This goal is achieved by defining several device classes in FTOS. To allow an easy expansion, FTOS uses the concept of class hierarchy supported by oAW. The basic sub

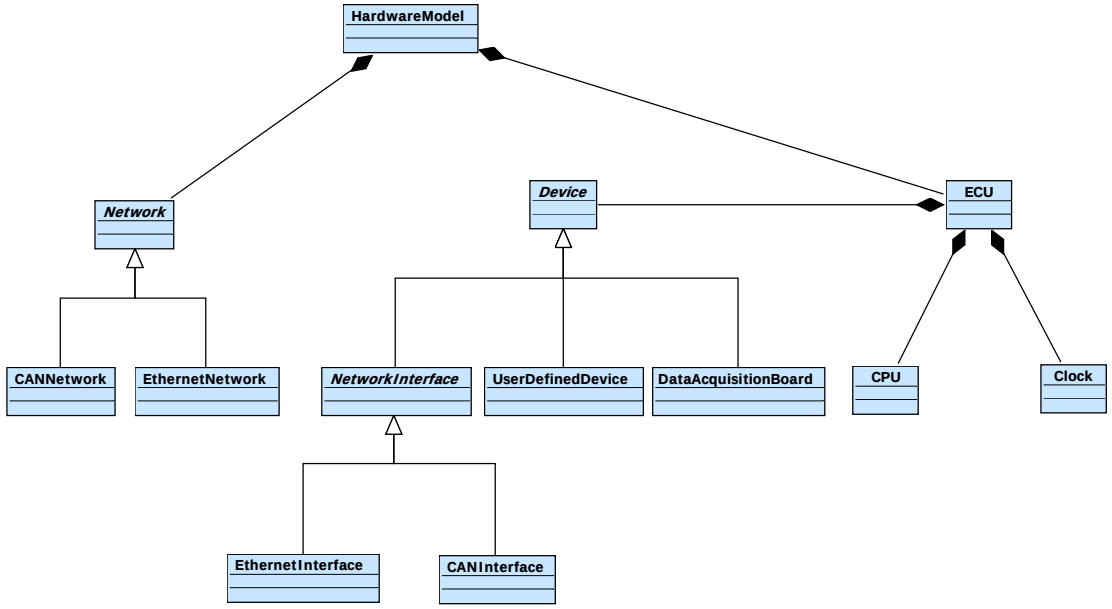


Figure 4.2: Hardware Meta-Model Structure

class for a device has the abstract syntax $(\texttt{d}) \in \texttt{Device}$ and describes only the unique name `d` of the device instance. Further sub classes are added and define the additional information required for automatic code generation. Examples for such device classes are `DataAcquisitionBoard` and `EthernetInterface`. The code synthesis is based on templates for each of these subclasses. The design of templates is simplified by the support of polymorphism in oAW. In addition, FTOS provides the sub class `UserDefinedDevice` to allow the use of new devices without having to expand the code generator and modeling tool. The abstract syntax for an instance is $(\texttt{d}, \texttt{init}, \texttt{delete}) \in \texttt{UserDefinedDevice}$ and refers in addition to the inherited name `d` to two functions `init` and `delete` that have to be implemented by the developer to initialize and delete the device.

Clock Synchronization: Section 4.4 presents the used model of computation, the concept of **logical execution times**. This concept requires a global time basis. Therefore, clock synchronization must be realized. The necessary information for the clock synchronization is contained in the definition of the clock components. The abstract syntax for a clock definition is $(\texttt{c}, \texttt{ticks}, \texttt{role}, \texttt{drift}, \epsilon) \in \texttt{Clock}$ and defines an unique name `c`, the clock rate `ticks`, the upper bound for the drift rate `drift`, the upper bound for the synchronization error ϵ, and the role within the synchronization `role`. The role defines whether the value of the clock is used for the clock synchronization. The concrete algorithm is described in Chapter 5.

Network: The definition of the network topology, the used network protocols, and other information is necessary for a correct implementation of mechanisms realizing the communication in the distributed system. FTOS uses again a class hierarchy to allow an easy expansion regarding additional network protocols. Figure 4.3 shows the concepts in the context of Ethernet networks. The model forces the developer to

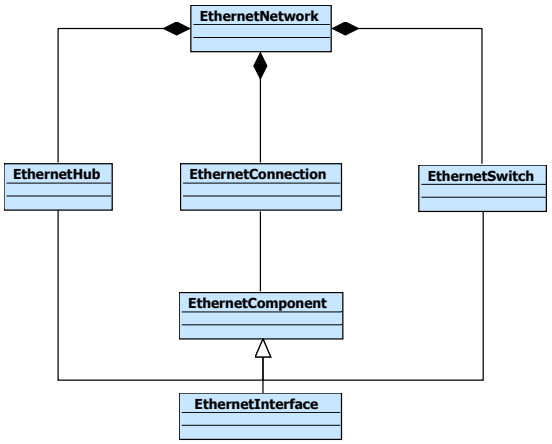


Figure 4.3: UML Model of the Network Aspects in the Hardware Meta-Model (simplified)

specify the exact topology by defining point-to-point links. This is very important in the context of fault-tolerant systems, since the different components of the network might be affected by errors. Only with knowledge about the exact topology, the effects of erroneous components can be determined exactly. The protocol specific classes contain additional information such as IP addresses for Ethernet or message identifiers for the Controller Area Network (CAN) protocol.

4.3.2 Hardware Model for Example Applications

The definition of the hardware model is straightforward. The TMR application controlling the rod consists of three ECUs that are connected by Switched Ethernet. Each unit is a standard desktop computer with an INTEL Pentium as processor. VxWorks 6.3 from Wind River[1] is selected as the operating system and C as the programming language. The interaction of the units with the environment is realized by data acquisition boards. Each unit has its own board, a ME2600 from Meilhaus, to measure the current position of the rod and output the result. The concrete model for the TMR application is depicted in Figure 4.4.

The model of the elevator application is more complex due to the higher number of control units that are used in the system. The control logic is realized by two standard desktop computers equipped with an INTEL Pentium Processor. Similar to the TMR application example, these units are running VxWorks 6.3. Besides the two control units, six ECUs are used to realize the interaction with the elevator hardware. On each floor, a board equipped with an AT90CAN128 [Atm07] processor performs the I/O operations. No operating system is used; the run-time system provides all necessary functionality. All units are connected via a CAN network.

[1] http://www.windriver.com/

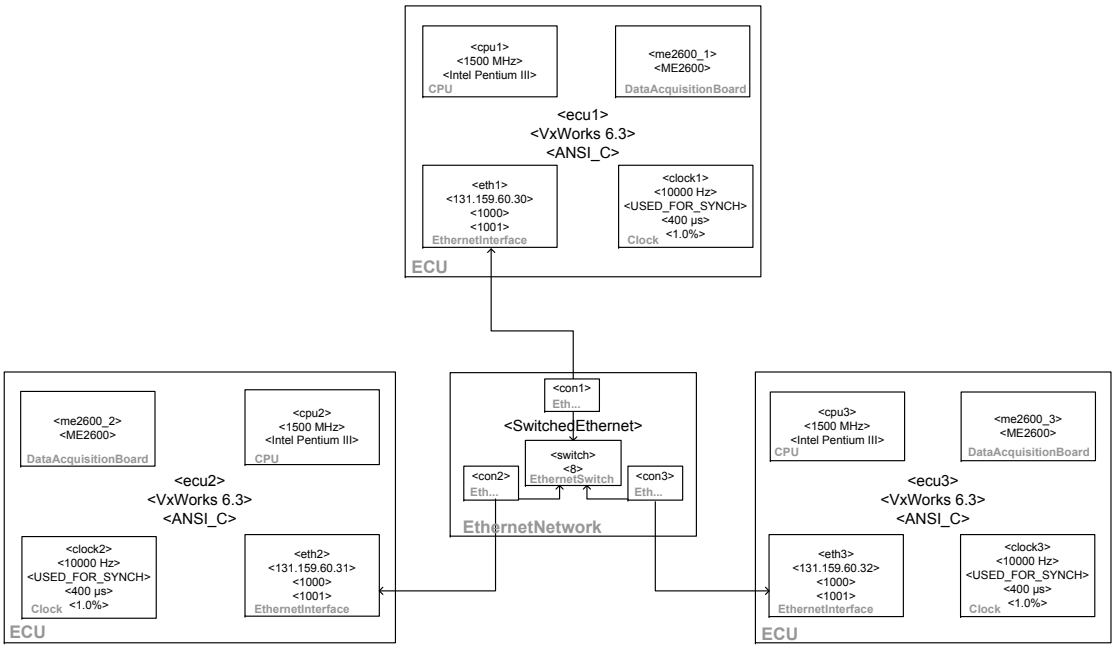


Figure 4.4: Graphical Representation of the Hardware Model

4.4 Software Model

The software model and especially its model of computation play a central role in FTOS. It is evident that the formulation of an adequate model can drastically simplify the implementation of the run-time system. This fact is also reflected in the requirements, where a majority targets issues related to the software model. The focus of the software meta-model is therefore to simplify the design of concurrent real-time systems and to be suited as foundation for the addition of fault-tolerance mechanisms. Standard approaches to design real-time systems treat timing and parallelism only in an indirect way and try to solve the problems with low-level constructs such as threads, priorities and semaphores [Lee06, GHI+06]. Even experts feel that the design of real-time systems becomes too complicated when using low-level constructs:

> *I believed that fine-grained concurrency was much too difficult for me. In my arrogance I thought that others might have difficulties as well. (Tony Hoare - Laser Summer School 2007)*

By raising the abstraction level from fine-grained concurrency to coarse-grained concurrency, this problem can be mitigated. In the following, the different concepts and design decisions are explained.

4.4.1 Main Concepts

This section presents the main concepts of the software model. Different projects were analyzed to gather existing solutions. Ptolemy [Lee03] provides a good overview on possible models of computation. Especially, Henzinger's project Giotto [HHK03] in-

fluenced the design of the meta-model. There are several follow-up projects of Giotto amongst others TDL [FFPT05] and HTL [GHI+06]. In addition, several fault-tolerance frameworks, e.g. DEAR-COTS [VCP+00] and TTA [KB03], were analyzed. Appendix A describes this related work for interested readers. The main concepts of the resulting software meta-model are listed below and are explained in more detail in the following.

1. Actor-Oriented Design: The application is interpreted as a set of concurrent software components. Actors reflect the application functionality.
2. Global Ports: Ports are used as communication points and are defined globally. They reflect the state of the system. The run-time system implements the necessary communication in the distributed system including consensus protocols.
3. Concept of Logical Execution Time (LET): A time-triggered approach is chosen as model of computation. The developer has to specify the logical start and stop time of the actors. The physical execution is realized by the scheduling component.
4. Mode Management: the developer can specify different schedules reflecting the possible application modes. The current schedule that should be executed is determined by a function, which is modeled by a finite state machine. The run-time system guarantees the consistent selection of the schedule within the distributed system.

Actor-Oriented Design

The main task of the software model is the identification of the software components that realize the application functionality and their interaction. The actor-oriented design by Agha [Agh86, AMST97] describes a system as a set of inherently concurrent software components and is therefore a good starting point. In contrast to Agha's work, FTOS restricts the communication concept between actors. While Agha's actors have an independent thread of control and use asynchronous message passing for communication, FTOS uses ports that are explained in more detail in the next section. The communication between the different actors is strongly restricted to guarantee determinism. Furthermore, actors in FTOS do not necessarily have to have an own thread of control. Only components with non-negligible execution times can be (depending on the implementation of the run-time system) executed within an own thread. These components are called **Tasks**.

Task: The tasks implement the actual application functionality. The abstract syntax for defining a task is $(\texttt{t}, \texttt{function}, \texttt{Reads}, \texttt{Writes}, \texttt{cpu}, \texttt{wcet}) \in \texttt{Task}$. Each task is identified by a unique name `t`. The attribute `function` refers to the task function that has to be implemented by the developer. It maps the values of the inputs $\texttt{Reads} \subseteq \texttt{Port}$ to the values of the output $\texttt{Writes} \subseteq \texttt{Port}$. Besides the platform independent information, the developer has to map the actors to the relevant hardware devices. The task mapping is specified by the attribute $\texttt{cpu} \in \texttt{CPU}$. In addition to the mapping, the developer can state the worst case execution time (WCET) [Wil03] $\texttt{wcet} \in \mathbb{Q}$. The specification of the

WCET is optional, but recommended if more than one task is executed in parallel on one ECU, and used by FTOS to realize a fault-tolerant scheduling scheme.

Two assumptions are made on the task function:

1. Simple Task Model [Kop97]: functions consist of sequential code without synchronization points.
2. Referential Transparency: functions are state-less.

The first requirement has two consequences: by banning synchronization points, problems such as priority inversion [LR80] and race conditions are excluded by design. This minimizes the implementation complexity and allows the implementation using platform independent languages. Therefore, there are no constraints regarding the programming language in contrast to other approaches, such as DEAR-COTS.

The second requirement can be motivated by the strict separation of the state from the functional behavior. All information reflecting the system state must be available to guarantee a correct implementation of voting and state synchronization protocols.

The two requirements enable FTOS to consider each function as black box. FTOS generates a stub that is responsible for the synchronization with the run-time system based on the task definition. The developer can therefore implement the task function without considering details about the applied fault-tolerance mechanisms, the used hardware or operating system. Thus, the intended goal of a two-phased design methodology is achieved.

Input and Output: The components realizing the interaction with the environment are called **Input** and **Output** actors. In contrast to approaches such as Giotto and HTL, the code for these components is generated by FTOS and has not to be implemented by the developer of the system. The execution of input and output actors is assumed to be instantaneous, similar to the synchrony assumption in synchronous languages [BCE+03]. To legitimate this assumption, the execution time must be negligible. Since different information might be necessary for code generation, FTOS uses inheritance and sub-classes for the definition of the different input and output types analog to the concepts used for the device definition.

The abstract syntax is $(\texttt{i}, \texttt{Writes}, \texttt{sensor}) \in \texttt{Input}$ for a basic input declaration and $(\texttt{o}, \texttt{Reads}, \texttt{actuator}) \in \texttt{Output}$ for a basic output declaration . In both cases, the developer has to specify a unique name `i` and `o`, respectively. The result of an input invocation is written to $\texttt{Writes} \subseteq \texttt{Port}$, the output is performed using the values of the port set $\texttt{Reads} \subseteq \texttt{Port}$. Similar to tasks, the input and output objects have to be mapped to the according hardware device, denoted by $\texttt{sensor} \in \texttt{Device}$ respectively $\texttt{actuator} \in \texttt{Device}$. Since the code is generated automatically, no function or information about worst case execution time is required.

Several additional properties can be defined for the individual sub-classes. The sub class `AnalogInput` augments the base class for example with an additional property `number`. By `number` the developer can specify the analog in-port of the referenced data acquisition board that should be used. Analog to the device class hierarchy, FTOS provides two sub classes that allow the definition of user defined input / output actors.

Global Port

Ports are used for the communication between the different actors. Several concepts of FTOS are achieved by ports:

1. Deterministic Access: The access on ports is performed time-triggered. If write accesses on one port are scheduled, the developer has to identify a unification strategy.
2. Reflection of System State: the values of ports represent the state of the system and can be used for voting and state synchronization protocols.
3. Abstraction of Communication: the communication within the distributed system is abstracted by the concept of ports.
4. Support of Heterogeneous Platforms: actors executed on different platforms (hardware, operating system, programming language) are able to communicate via ports. The conversion is realized by the run-time system.

The abstract syntax for a port declaration is $(\texttt{p}, \texttt{type}, \texttt{init}, \texttt{rep}, \texttt{unify}, \texttt{const}) \in \texttt{Port}$ and consists of a unique name `p`, the platform-independent type of the port `type`, an initial value `init`, a Boolean value denoting the replication type `rep`, the unifying strategy `unify` and a Boolean value `const` denoting whether the port is a constant.

To guarantee a deterministic execution, the actors can access the port only at their logical start and end. For input and output actors the implementation is trivial, as the execution is assumed to be instantaneous. In case of tasks, the read operations are performed at the logical start of a task by copying the values of the ports into the memory of the task. At the logical end of a task, the write operations are performed by copying the results from the task's memory into the relevant ports. For this purpose, **private ports** $\texttt{P}_\texttt{t}$ are created for a task `t` within the generated task stub. This approach avoids the necessity for sophisticated data objects as required in other approaches, e.g. DEAR-COTS. Simultaneous write accesses on a single port must of course be handled by the system to guarantee consistency. While simultaneous write accesses are forbidden for example in HTL, they are very common in fault-tolerant systems due to actor replication. FTOS offers two possibilities to reach consistency: unifying strategies and port replication. Unifying strategies define a precise action to calculate a unique value out of redundant results, analog to composite operators in Esterel [BG92]. Possible strategies are the calculation of an average value, the selection of the median or an arbitrary value. The idea behind port replication is explained below.

The values of ports reflect the state of the system. This is the reason for the difference between the port concept of FTOS and the concept used in Giotto and TDL. In the latter approaches, a port is assigned to a specific actor and the communication is realized by communication channels. However, the approach of using local ports is not suitable for fault-tolerant systems. Figure 4.5 shows an example that gives an intuitive reason. Three tasks are replicated to achieve fault-tolerance. An output actor uses the result of the redundant tasks. However, it is not obvious which port should be read by the output actor. The concepts of global ports and of unification strategies solve this dilemma. Similar to HTL and its concept of **communicators** and **task ports**, a port is a discrete component. The values of the declared ports reflect the state of the system (besides the

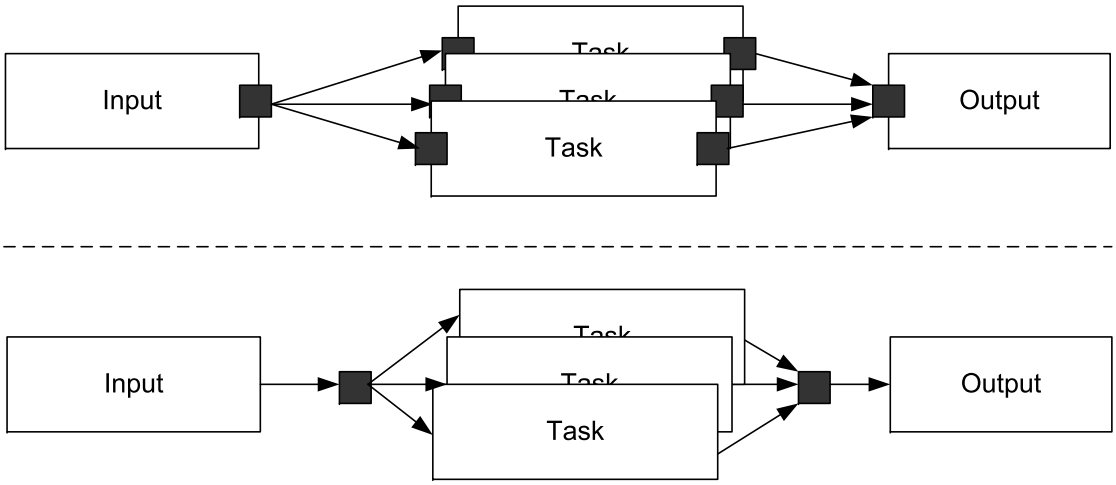


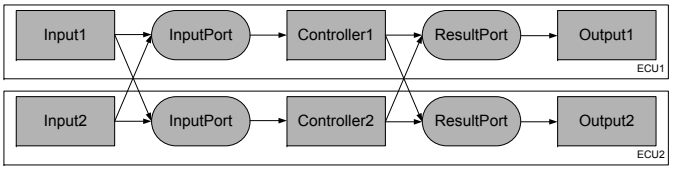
Figure 4.5: Local Ports vs. Global Ports

current logical time and the set of active modes). Port values are in general consistent within the distributed system.

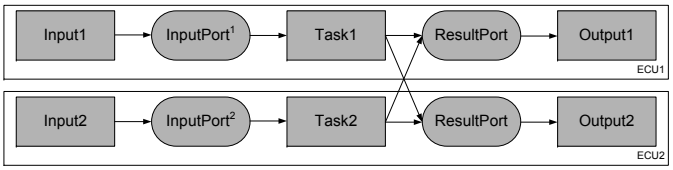
The communication is realized by the concept of ports in a transparent way, analog to the shared data objects in combination with the communication manager in DEAR-COTS. The developer can simply define two actors running on different hosts that communicate via ports. The communication is realized by the run-time system. This realization is complicated by the possibility of faults. Therefore, consensus protocols are required to guarantee a consistent execution within the distributed system. These protocols might require several communication rounds. One advantage of the time-triggered approach is that the number of required rounds is bounded. The concrete number of rounds depends on the fault assumptions. To lower the number of communication rounds, FTOS offers possibilities to differentiate between the necessity to reach a strictly consistent value and a steady, but probably imprecise adaptation. For this purpose, FTOS offers two possibilities for each unification strategy, see Figure 4.6. `AVERAGE_CONCENSUS` is used to calculate a consistent average value out of all correct results. In contrast, the strategy `AVERAGE` is used to calculate similarly an average value out of the correct results, but due to message loss, the individual set of ECUs may not agree on this set and might have a slightly different result. The developer can decide which of the different possibilities meets the requirements of the application.

However, even the reduced complexity of the imprecise protocols might not be necessary when redundant actors are reading the results from yet other redundant actors. Figure 4.7 gives an example. Instead of performing a voting round both after the execution of the input actors and after the execution of the task function, it might be useful to perform only one voting round after the task execution. One possibility to avoid the first consensus round is to specify unique ports for each of the redundant branches. However, this is not practicable because the ports have nevertheless a semantic relationship that might be exploited for example during state synchronization. FTOS solves this problem by introducing local port replication. A replicated port be-

Figure 4.6: Unifying Strategy for Ports



(a) Consensus After Each Step with Standard Ports



(b) Reduced Communication Rounds with Replicated Ports

Figure 4.7: Standard Ports vs. Replicated Ports

haves similar to a standard port, but only local access operations are performed. No communication or consensus execution is therefore necessary for this type of port. This approach resembles the concepts of internal outputs in DEAR-COTS.

Concept of Logical Execution Time

Henzinger [Hen04] emphasized the importance of an adequate model of computation to simplify the design and implementation of embedded software. Lee [Lee00] pointed out that the composition of components should have consistent and non-conflicting temporal properties. Especially in the context of fault-tolerant systems, where different experts are involved in the development of the system, a simple execution model is essential. Ptolemy provides a good overview on the different models of computation [Lee03]. An execution model that fulfils these requirements is the concept of logical execution times (LET) as proposed by Henzinger in the context of Giotto [HHK03]. Figure 4.8 points out the general concept. Instead of using the physical execution time, all computations logically take a fixed amount of time [Kir02]. The logical start and

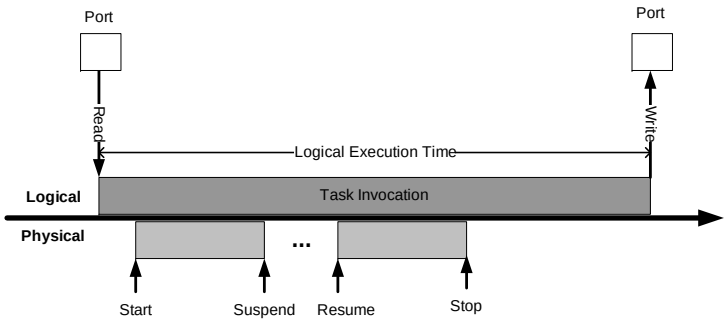


Figure 4.8: Concept of Logical Execution Times

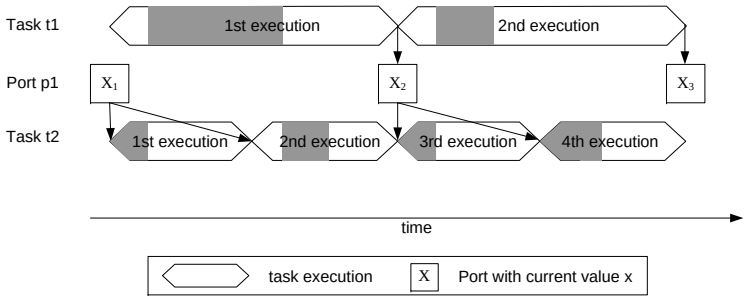


Figure 4.9: Deterministic Execution by Using Concept LET

stop times are specified by the developer within a schedule. At the logical start of a task $t \in$ `Task`, the values of the variables `t.Reads` $\subseteq$ `Port` are copied into the task's memory. The results are copied back into the variables of `t.Writes` $\subseteq$ `Port` at the logical end of task `t`. The physical execution of the task on the CPU becomes transparent to the application developer and is managed by the run-time system.

The major advantage of logical execution time is the existence of previously known points in time, when fault-tolerance mechanisms are executed. Using the time-triggered interaction scheme between the different components, message loss or additional messages are directly discovered. The implementation of fault-tolerance mechanisms is therefore simplified.

In addition, there are several other advantages: problems such as race conditions are excluded by design and flexibility with respect to replica diversity is guaranteed. Figure 4.9 explains why race conditions are excluded: independent of the actual time when task `t1` finishes its computation, task `t2` will only read the new result every odd execution. Another important advantage of the concept of logical execution time is the provided flexibility with respect to replica diversity. Since the physical execution is abstracted, the tasks must only behave similarly at the logical level. This increases the flexibility with respect to replica determinism.

Mode Management

An application can have different execution modes. Each mode is associated with a schedule that defines the execution times for various actors. In general, the system executes the schedule of the current mode exactly once (one mode cycle) and determines the next mode afterwards. By switching the mode, the schedule may change and also different actors may be executed. The main concepts of FTOS to support the mode management are:

1. Use of State Machines: state machines are used to model mode switch functions
2. Distributed Mode Management: the run-time system guarantees a consistent mode decision on all correct nodes
3. Job Concept: several independent jobs can be executed within the system, each referring to its own mode
4. Introduction of Sequential Execution: instead of using only periodic executions of actors, FTOS allows the definition of arbitrary time-triggered schedules
5. Definition of Replication Types: the developer can easily specify components that are executed in passive, semi-active or active mode

Mode switches define the transitions between different modes. FTOS uses state machines to model the mode switch function instead of relying on functions implemented by the developer as it is for example the case in Giotto. This technique of combining different types of models of computation on different levels of abstraction, here logical execution times and finite state machines, is called **modal models** [Lee03]. The major benefit of the approach to use state machines in contrast to a function implemented by the developer is the availability of information that can be used for optimization.

The use of modes is restricted in most other approaches: Giotto allows only exactly one mode to be active at a time. However, many control applications perform independent application missions at a time. Each mission might have different operational modes that are independent from the modes of other missions. By insisting on the limitation of at most one active mode, for each possible combination of mission modes, an application mode has to be defined. This leads to an exponential increase in the number of modes. Furthermore, non-harmonic mode switches have to be introduced to switch the mode during a mode cycle. TDL and HTL introduce modules to solve this problem. Modules are a set of actors and a set of modes. The modes between different modules are independent. However, the modules are restricted in both approaches in their physical dimension. Only actors executed on one ECU can be contained in one module. Distributed mode switches have to be realized by the application developer himself. This is acceptable for most applications where mode switches are rather rare. In fault-tolerant systems however, the importance of different execution modes is much higher [RC04]. Besides the application modes, a variety of administrative modes, such as fault-recovery or emergency mode, have to be implemented in fault-tolerant applications. Due to the distributed nature of fault-tolerant systems, the different units must perform consistent mode switches even in the presence of faults. Therefore, the realization of distributed fault-tolerance mechanisms is a necessity for FTOS.

To cope with this problem, FTOS introduces the concept of a job. A job is similar to the modules of TDL and HTL a set of modes. Each mode references a schedule. However, the set of referred actors is not restricted to actors executed on only one ECU. The abstract syntax for a job is $(\texttt{j}, \texttt{start}, \texttt{Modes}, \texttt{change}, \texttt{idleTime}) \in \texttt{Job}$. A job with a unique name `j` can consist of several modes $\texttt{modes} \subseteq \texttt{Mode}$, where $\texttt{start} \in \texttt{Modes}$ is the start mode of the job. The state machine $\texttt{change} \in \texttt{StateMachine}$ evaluates potential mode changes. The decision can be based on port values and ports can also be written to implement for instance a counter functionality. The state machine may also determine that currently no mode should be executed. The transitions of a state machine can be specified in a platform independent way using a notation similar to LaTeX. The time interval until the next evaluation is denoted by `idleTime`. A mode declaration $(\texttt{m}, \texttt{schedule}, \texttt{duration})$ defines a unique name `n`, the according schedule $\texttt{schedule} \in \texttt{Schedule}$ and the duration of one mode cycle `duration`. Besides application modes, a job may also contain administrative modes. These modes are not reached by a standard state transition, but rather invoked as reactions to detected errors.

A schedule defines the execution time for different actors. Instead of relying only on periodic executions, it can be useful to have sequential executions. A good example is a task performing some preprocessing before sending some values to a task on another ECU to save bandwidth. FTOS provides three different types of schedules that are defined as sub classes of the Schedule. The class Schedule is again a sub class of ScheduleReference to allow a recursive definition of schedules.

The declaration of a StandardExecution schedule has the abstract syntax $(\texttt{s}, \texttt{Item}) \in \texttt{StandardExecution}$ and defines a unique name `s` and a schedule list $\texttt{Item} \subset \texttt{ScheduleReference}$. All items in the list are executed.

In contrast, an AlternativeExecution schedule allows the definition of a list of schedules where at most one schedule is executed at a time. The abstract syntax is $(\texttt{s}, \texttt{Schedules}, \texttt{trigger}) \in \texttt{AlternativeExecution}$. The trigger object $\texttt{trigger} \in \texttt{Trigger}$ evaluates which schedule $\texttt{sched} \in \texttt{Schedules}$ is executed. The definition of a trigger function is similar to the definition of state transitions. The function can read and also write ports.

The third option is the definition of parallel executions that are especially useful to describe the execution of replicas. The abstract syntax for a parallel execution object is $(\texttt{s}, \texttt{type}, \texttt{Replica}) \in \texttt{ParallelExecution}$. The schedules referenced by $\texttt{Replica} \subset \texttt{Schedule}$ are executed according to the replication type. Three main replication approaches are addressed in the literature [PCD91]:

- **Passive replication**: only one replica is executed at a time. If an error is detected, the active replica is excluded and a spare replica is activated. The approach is also known as **cold redundancy** or **primary backup**.
- **Active replication**: all replicas are executed at a time. The approach is also known as **hot redundancy**.
- **Semi-active replication**: similar to active replication all replicas are executed except the output operations. A master is selected that performs the output exclusively.

Property	Value
Cpu	CPU CPU1
Function	control
Name	PIDController1
Reads	Port Measurand, Port ErrorSum, Port LastValue
Wcet Nanoseconds	-1
Wcet Seconds	-1
Writes	Port ErrorSum, Port LastValue, Port Result

Declaration of Task PIDController1

Property	Value
Name	In1
Number	1
Sensor	Data Acquisition Board ME2600_1
Writes	Port Measurand

Declaration of Input In1 (Class: AnalogInput)

Property	Value
Actuator	Data Acquisition Board ME2600_1
Name	Out1
Number	0
Reads	Port Result

Declaration of Output Out1 (Class: AnalogOutput)

Figure 4.10: Examples of Actors in TMR Control System

All these different possibilities can be specified directly in FTOS by the redundancy type `type`. Possible values are `None`, `ColdRedundancy` for passive replication, `HotRedundancySingleOutput` for semi-active replication and `HotRedundancyMultipleOutput` for active replication [2].

The different types of schedules can be nested arbitrarily. In addition to other schedules, StandardExecution schedules can also contain actor references. For each type of actor, an own sub class is defined. The abstract syntax of the abstract base class ActorReference, a sub class of ScheduleReference, is $(\texttt{actor}, \texttt{start}, \texttt{guard}) \in \texttt{ActorReference}$ and defines the logical start time $\texttt{start} \in \mathbb{Q}$, the relevant actor `actor` and a trigger $\texttt{guard} \in \texttt{Trigger}$ that evaluates whether the item should be executed. The class TaskReference adds an additional attribute $\texttt{stop} \in \mathbb{Q}$ that specifies the logical stop time.

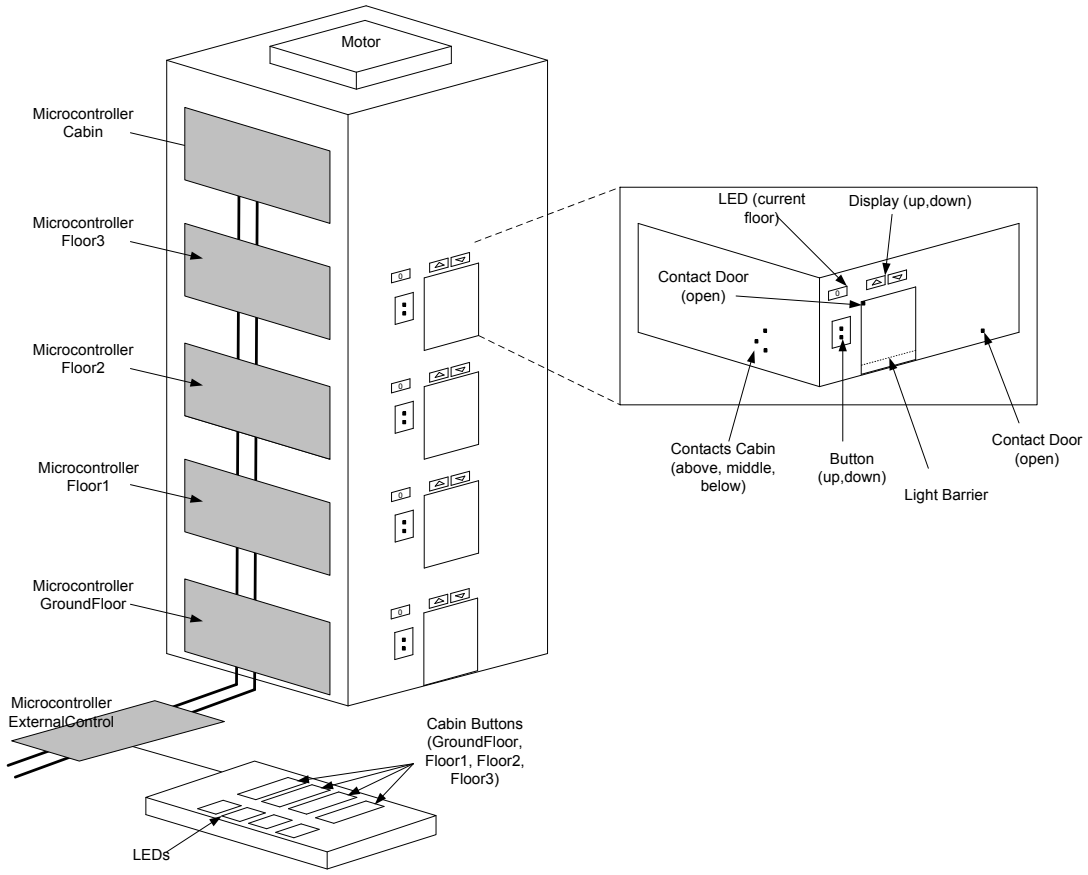


Figure 4.11: Elevator

4.4.2 Software Models of the Demonstrators

Actors of the Balanced Rod and Elevator Control System

The definition of the required actors is straightforward. In the TMR example, the PID control function is replicated on three redundant control units. Therefore, three tasks `PID_CONTROLLER1`, `PID_CONTROLLER2`, and `PID_CONTROLLER3` are defined. Each task is assigned to one CPU.

For the interaction with the environment, actors to measure the current position of the rod and to control the voltage of the solenoids have to be defined. These actors are based on the data acquisition boards specified in the hardware model. Three `AnalogInput` actors and three `AnalogOutput` actors are defined to realize the measurement respectively the control. The models for a task, an input and an output actor are depicted in Figure 4.10.

The elevator example is realized analog to the TMR example. For each input and output operation a discrete actor is specified. The elevator setup is depicted in Figure 4.11.

[2]These differing terms are used due to the fact that the definition of semi-active and active replication in the literature is inconsistent.

On each floor there are two sensors to detect the status of the door (open, close), a light barrier to detect persons at the sill, two buttons to request the elevator and three contacts to determine the position of the cabin. In addition, the current floor of the cabin can be displayed by seven-segment display and the moving direction of the elevator is indicated by two LEDs. Summarily, each microcontroller controlling a floor has eight sensors and 3 actuators. The micro controller controlling the cabin has one actuator. Using an additional micro controller, the panel inside the cabin is simulated. Four buttons are available for the individual floors and for each floor a LED displays whether a request is pending. Each of the sensors and actuators is represented by an actor. Therefore, the model contains 36 input actors and 17 output actors for each elevator. Two redundant tasks implement the control function. In addition, six task functions are defined for each micro controller. These tasks perform the emergency mode, if both control units fail.

Ports

For the TMR control system, it is necessary to analyze the control function in more detail. The tasks implement a proportional-integral-derivative (PID) controller. A PID control function tries to minimize the deviation $\mathtt{e(t)}$ between a measured process variable and a desired set point. As the name implies, the PID controller consists of three terms. The proportional term $\mathtt{K_p} \cdot \mathtt{e(t)}$ changes the output proportional to the current error. The integral term $\mathtt{K_i} \cdot \int_0^{\mathtt{t}} \mathtt{e}(\tau)\mathtt{d}\tau$ helps to eliminate steady-state errors. To improve the stability of the control process, an integral term $\mathtt{K_d} \cdot \mathtt{de(t)/dt}$ is used. Altogether, the PID controller task function implements the following function (discretized version of PID controller):

$$f(t_i) = K_p \cdot e(t_i) + K_i \cdot \sum_{\tau=0}^{i} e(\tau) + K_d \cdot \big(e(t_i) - e(t_{i-1})\big) \tag{4.1}$$

The results of the input actors are written to a port `InputPort`. To calculate the error $\mathtt{e(t)}$, the set point can be realized as a constant port `SetPoint`. Alternatively, the set point can be hard-coded in the function. To calculate the derivative and integral terms, additional ports are necessary, because internal variables are not allowed. These ports `ErrorSum` and `LastValue` are required to calculate the error sum and the difference between the current position and the previous position. The result of the PID controller function is written to the port `Result`, which is read by the output actor. The data flow for one control unit is depicted in Figure 4.12.

The next step is to select an appropriate mechanism to cope with multiple write accesses. To minimize the communication overhead, it is useful to restrict the number of transmitted variables and the number of communication rounds. The first approach is certainly to replicate the port `InputPort` locally on each ECU to avoid the first consensus round. The port `Result` must be definitely used during voting. Interestingly, this is also true for `ErrorSum`. This can be motivated by analyzing the PID function. Within the port `ErrorSum`, the sum of the measured error is stored. The calculation of this error is based on the result of an analog input. Due to the impossibility of a perfect sensor calibration, there will always be a small deviation in the computed error of the differ-

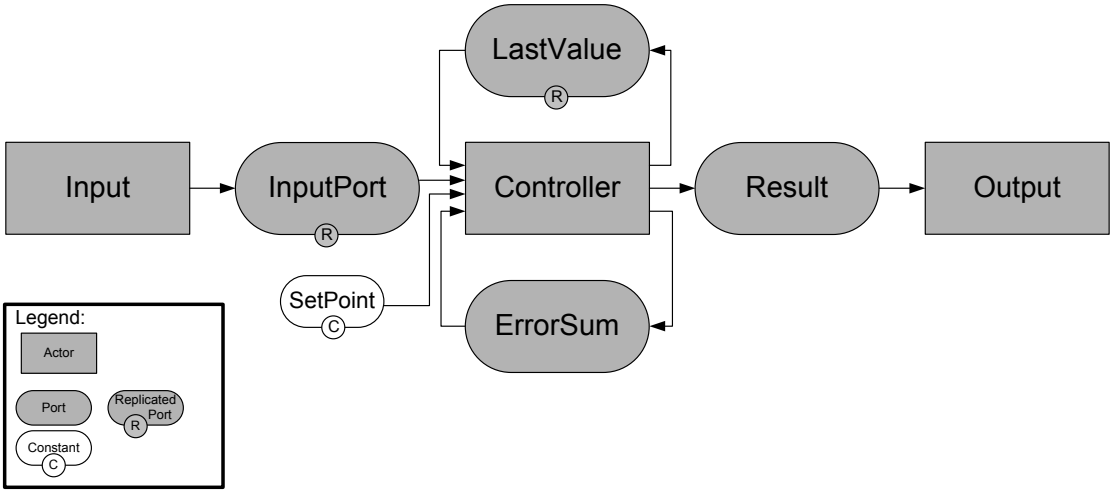


Figure 4.12: Software Model Example: PID Control Application

ent units. This seems at first glance acceptable, since a small deviation on the output does not harm the system. However, this small deviation will quickly sum up in the value `ErrorSum`. Finally, it will lead to the fact that the results of the three replicated units drift from each other unboundedly. Therefore, the port `ErrorSum` must be also considered during the consensus operation. Because small differences are acceptable, it is sufficient to select the calculation of an average value without consensus as the unification strategy.

The port concept allows a simple definition of the required variables in the elevator model. For each input and output actor, an according port is specified. The control function operates on these ports and uses an additional port to store the current state of the elevator (position, moving direction). This additional port needs not to be compared, as only fail-silent behavior is expected. For all other ports, an arbitrary unifying strategy can be selected.

Jobs, Modes and Schedules

For the TMR control, only one job is specified. The job defines one mode that controls the rod. The schedule of this mode refers to a `ParallelExecution` schedule that consists of three redundant schedules. These schedules are operated using semi-active replication. In each redundant schedule, the timing of the individual item is specified. At logical time `0ms` (relative to the begin of the mode cycle) the input is executed. The according task is logically also started at time `0ms`. The logical end of the task execution is `2.5ms`. At the same logical moment, the analog output is performed.

The elevator control is more interesting. In principle, there are four different modes of operation: initialization (start mode), normal operation and two emergency modes. During initialization the system transfers the elevator to an initial position (e.g. moving the elevator to the ground floor). The normal operation mode provides the expected elevator service. The emergency modes transfer the elevator into a fail-safe position

(e.g. moving the elevator to the next level) and then stop the service afterwards. This operation is conducted either by the control units, if a field controller fails, or by the field controllers if the two control units fail.

Nevertheless, it is not useful to map each of the modes of operation to an individual mode. In general, modes should only be used if different sets of actors are involved and/or the schedule changes. A reason for this recommendation is the overhead of modes. As all participating nodes have to agree on the current executed task, the relevant information must be available on and processed by each node. In the context of the elevator application, this would imply that every controller would have to monitor every other node. However, it is obvious that only the control units should monitor all field controllers and the other control unit, while the field controllers only need to monitor the two control units. If only one actor function should be changed, as it is the case in the elevator example, the change should be realized within the application logic. FTOS supports this application-dependent specification by several means that are discussed in Section 4.6.

Summarily, it is good practice to realize the elevator control with only two modes: one mode combining the initialization, normal operation, and emergency mode conducted by the control units and the other implementing the emergency mode conducted by the different field controllers. Figure 4.13 depicts the two modes including the submodes realized in application logic. No transition between the two modes is specified in the software model, as the transition is triggered by the detection of errors. These fault-tolerance related transitions are specified in the fault-tolerance model.

The schedule for the first mode consists of the execution of all input actors at time `0ms`. Subsequently, the two control tasks are executed in parallel execution on the control units. As replication paradigm, active replication is chosen. The logical execution is performed from `0ms` to `50ms`. Finally, the output operations are performed at time `50ms`. The second task is defined analog to the first mode, but replaces the tasks of the control units by the spare tasks executed on the field controllers.

4.4.3 Execution Order of Logically Simultaneous Actions

Using the concept of logical execution time, it is common to specify several actions to be performed at the same logical moment. However, the execution order of the different actions is important for the correctness of the system. An example is the TMR application. The task should read the sensor result at the logical time `0ms` and therefore start its execution after the input function is executed. A similar condition can be stated for the logical time `2.5ms`. First, the task should publish its result and afterwards the output should be executed. FTOS uses the concept of **micro time** to reflect this execution order. The micro time specifies for each type of action, when the action is executed in comparison to actions of another type. Actions of the same type can be executed in arbitrary order. The `microTime` attribute is added to the ScheduleReference class during M2M transformation.

Besides starting and stopping tasks and executing input and output actors, the run-time system must perform several other actions. The results of the individual actors might

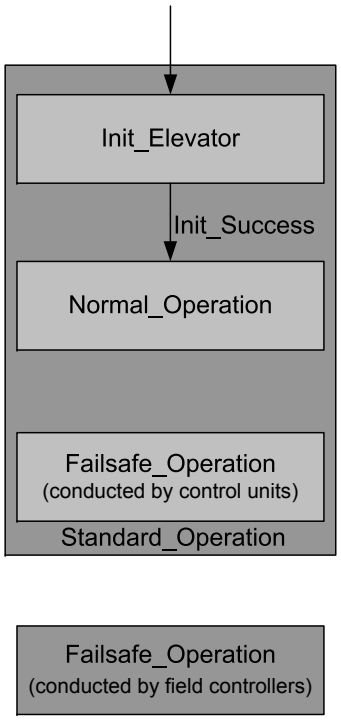


Figure 4.13: Elevator - Modes and Transitions including Sub-Modes

need to be transferred to other ECUs and unifying operators must handle redundant results. In addition, triggers need to be executed to evaluate whether an actor or a complete sequence should be executed. These triggers may be executed at different positions within a logical moment. Triggers associated with a concrete actor (guard) are evaluated directly before the execution of the according actor types. Triggers associated with a complete sequence are executed once at the beginning of this sequence, meaning before the earliest element of the sequence. The exact position is computed during M2M transformation, which is explained in detail in the next section. Figure 4.14 shows the different actions and their order. Single actions, such as the logical end of a task, are split up into different steps. In the following, these steps are explained in more detail:

1. Task Completion: The first steps at a logical moment are used to terminate the execution of tasks that logically complete in this moment. If the task could not complete its physical execution on time, an error flag is raised (see fault-tolerance mechanisms). If the task was able to complete the execution, send operations are executed to propagate the result, if necessary. Finally, the task results are copied into the global ports taking into account potential unifying operators.
2. Trigger Execution (before Output): Triggers associated with an output element are executed next. The according ports are read, the trigger function is executed and results concerning ports (if any) are written. The write access of a trigger is limited to a port that is locally replicated and exclusively accessed by this trigger.

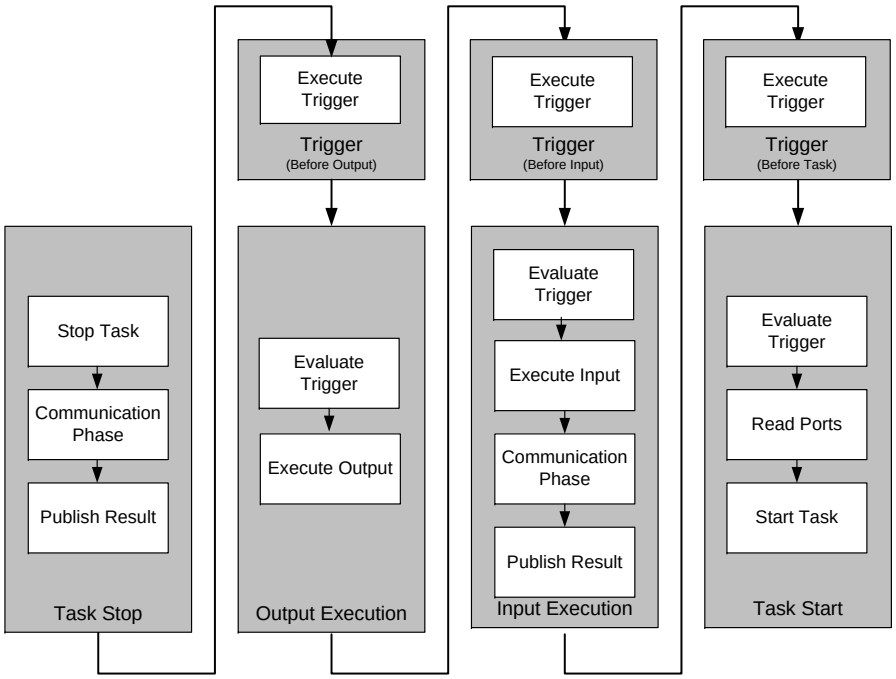


Figure 4.14: Execution Order of Logically Simultaneous Actions

The port therefore serves as private variable. This constraint is used to avoid additional communication rounds and is tested during model validation.

3. Output Execution: The execution of output actors is the next phase at a logical moment. For each output actor, the relevant trigger results are evaluated. Only if all associated triggers signal a positive evaluation, the actor execution is performed. For all positively assessed actors, the associated port values are read and the output function is executed. This execution is performed synchronously within the systems execution. The execution of the output actors completes logically a round in the sense of Giotto.
4. Trigger Execution (before Input): At the begin of the next round, triggers are again executed. During this phase, all triggers associated with input actors that are scheduled for this logical moment are evaluated.
5. Input Execution: Similar to the output phase, the input execution starts with an evaluation of the associated triggers. Afterwards, the positively assessed input actors are executed. A communication phase realizes the propagation of the result, if necessary. The input results are then published (copied into the according ports) taking into account potential unifying operators.
6. Trigger Execution (before Task): After the input execution, the third trigger phase is carried out. During this phase, all triggers associated with tasks that are scheduled for this logical moment are executed.
7. Task Start: The last phase at a moment is dedicated to the logical start of tasks. For all task references with a start time that matches the logical time, the trigger

outcomes are evaluated. For all positively assessed ports, the values of the ports in the `Read` set are copied into the private ports of the task and the task reference is passed to the scheduler to realize the physical execution.

Using this execution order, the system can be built straightforward. However, due to the potentially nested schedules and some information that is only implicitly contained, e.g. trigger information, the specified schedule must be simplified and sequentialized. This simplification is done during the M2M transformation that is explained in the next section.

4.4.4 Model-To-Model Transformation in the Context of the Software Model

Especially the software model contains a lot of implicit information. On the one hand, this simplifies the modeling phase, but complicates the code generation on the other hand. The task of the Model-To-Model (M2M) transformation is to compute this information and to add this information to the combined model. One example is the concept of ports. Ports are defined globally without specifying which ECUs need to have a copy of the port object. During M2M transformation, the port class is augmented by two additional references $\texttt{ReadECUs} \subseteq \texttt{ECU}$ and $\texttt{WriteECUs} \subseteq \texttt{ECU}$ that refer to the ECUs executing actors that read respectively write the port value. Other examples are the expansion of the actor class with a reference $\texttt{ecu} \in \texttt{ECU}$ denoting the executing ECU and the expansion of the job class with a similar reference $\texttt{ECUs} \subseteq \texttt{ECU}$ denoting the ECUs that execute parts of the job. All references are of course bidirectional references. The ECU class is for example augmented with a reference $\texttt{Job} \subseteq \texttt{Job}$ denoting the executed jobs.

This section illustrates the concept of the M2M transformation using the calculation of an expanded schedule as an example. Besides the already contained explicit information about the execution times of the individual actors, the schedule should also contain information about

- Execution Conditions beyond Guards,
- Points in Time for Trigger Evaluation,
- Network Communication and
- Execution of Unifying Operators.

The goal of the M2M transformation is to produce for each mode a sequential schedule containing the information listed above. The model transformation is performed in several steps:

1. Resolve `AlternativeExecution` and `ParallelExecution` Schedules
2. Determine Points in Time for Trigger Execution
3. Determine Network Communication Elements
4. Determine Publish Objects
5. Calculate Local Schedules
6. Sort Schedule

In the following, the individual steps are explained in more detail. For simplification reasons, the system is assumed to contain exactly one mode.

Resolve AlternativeExecution and ParallelExecution Schedules

The first step to produce a sequential schedule is to resolve the specified `AlternativeExecution` and `ParallelExecution` schedules. The class `ActorReference` is augmented with a reference `TriggerList` that references trigger condition objects. The abstract syntax of a trigger condition instance is $(\texttt{trigger}, \texttt{value}) \in \texttt{TriggerConditions}$. The actor reference is only executed if the result of the referenced `trigger` matches the specified value `value` for all trigger conditions in `TriggerList`.

`TriggerList` can be calculated as follows. In the first step, each ParallelExecution object is augmented with a trigger `trigger` realizing the voting decision. The trigger functionality is generated during M2M transformation of the fault-tolerance model and specifies which replica (passive replication) or which replica's output elements (semi-active replication) is executed.

The second step computes all relevant AlternativeExecution $\texttt{relA}(\texttt{r})$ and ParallelExecution schedules $\texttt{relP}(\texttt{r})$ for a specific schedule reference `r`:

$$\begin{aligned} &\forall r \in (ScheduleReference) : \\ &\quad relA(r) = \{a \in AlternativeExecution \mid a.contains(r) = true\} \\ &\quad relP(r) = \{p \in ParallelExecution \mid p.contains(r) = true\} \end{aligned} \tag{4.2}$$

A trigger result refers always to the position of the elements in the associated ordered set that should be executed. The list of trigger conditions `r.TriggerList` for a schedule reference `r` is derived by the following equation:

$$\begin{aligned} &r.TriggerList = \\ &\quad \{(tr, v) \mid (\exists s \in relA : s.trigger = tr \wedge v = pos(s.schedules, r)) \vee \\ &\qquad ((\exists s \in relP : s.trigger = tr \wedge v = pos(s.schedules, r) \\ &\qquad\quad \wedge s.type(ColdRedundancy)) \vee \\ &\qquad ((\exists s \in relP : s.trigger = tr \wedge v = pos(s.schedules, r) \\ &\qquad\quad \wedge s.type(HotRedundancySingleOutput) \wedge r \in OutputReference) \vee \\ &\qquad (tr = r.guard \wedge r.guard \neq bot \wedge v = 0)\} \end{aligned} \tag{4.3}$$

By adding the trigger conditions directly to the individual items, it is possible to define a new schedule $\texttt{s} \in \texttt{StandardExecution}$ that contains all actor references, but no other schedule.

Determine Points in Time for Trigger Execution

The execution of all elements previously contained in an AlternativeExecution schedule `a`, depends on the result of the associated trigger. The trigger is evaluated in every

mode cycle exactly once before the execution of the first associated element. The exact execution time can be derived analyzing the associated elements. The first step is to find the set of earliest elements:

$$earliest(a) = \{e \mid e \in a.elem : \ \forall j \in a.elem : \ e.start \leq j.start\} \tag{4.4}$$

The logical point in time for the trigger execution `start` is defined by `get(earliest(a), 0).start`. The exact schedule position `microTime` of the trigger depends on the types of elements contained in `earliest(a)`:

$$microTime = \begin{cases} BeforeOutput & , \exists o \in OutputReference : earliest.contains(o) \\ BeforeInput & , \neg(\exists o \in OutputReference : earliest.contains(o)) \\ & \wedge \exists i \in InputReference : earliest.contains(i) \\ BeforeTask & , otherwise \end{cases} \tag{4.5}$$

For each trigger associated with an AlternativeExecution, an instance $(\texttt{trigger}, \texttt{start}, \texttt{microTime}, \texttt{ECU}) \in \texttt{TriggerReference}$ of the TriggerReference class is inserted in the schedule. `ECU` references all ECUs that contain actors that are influenced by the trigger decision. The computation is straightforward and omitted, similar to the addition of the trigger reference for guards.

Determine Network Communication Elements

The local results of an input or task execution may be required on another `ECU`. Therefore, additional communication objects $(\texttt{p}, \texttt{ar}) \in \texttt{CommunicationReference}$ have to be added to the schedule, where $\texttt{ar} \in \texttt{ActorReference}$ refers to the actor reference providing the result and $\texttt{p} \in \texttt{Port}$ refers to the written actor. The communication objects `CR` that need to be added to the schedule are derived by the following equation:

$$\begin{aligned} CR = \{(p, ar) \mid & \ p \in Ports \wedge ar \in (InputReference \cup TaskReference) \wedge \\ & \wedge \ !p.isReplicated \\ & \wedge \ p \in ar.actor.Writes \\ & \exists ecu \in p.ReadECUs \\ & \wedge \ ecu \neq ar.actor.ecu\} \end{aligned} \tag{4.6}$$

Determine Publish Objects

The last elements that need to be added to the schedule are publish objects. The abstract syntax of this object is $(\texttt{p}, \texttt{AR}) \in \texttt{Publish}$. The publish object realizes the update operation of the value of port `p` under consideration of the relevant unifying strategy. The set `AR` contains the according actor references that contribute relevant results. The

set of publish objects `Pub` is derived from the following equation:

$$\begin{aligned}
Pub = \{(p, IR) \mid\ & p \in Ports \wedge !p.isReplicated \wedge IR \subseteq InputReference \\
& \wedge (\forall i, i' \in IR : p \in i.actor.Writes \wedge i.start = i'.start) \\
& \wedge \neg(\exists i' \in InputReference \setminus IR, \exists i \in IR : i.start = i'.start \\
& \quad \wedge p \in i'.actor.Writes)\} \\
\cup \{(p, IR) \mid\ & p \in Ports \wedge p.isReplicated \wedge IR \subseteq InputReference \\
& \wedge (\forall i, i' \in TR : p \in i.actor.Writes \wedge i.start = i'.start \wedge i.ecu = i'.ecu) \\
& \wedge \neg(\exists i' \in InputReference \setminus IR, \exists i \in IR : i.start = i'.start \\
& \quad \wedge i.ecu = i'.ecu \wedge p \in i'.actor.Writes)\} \\
\cup \{(p, TR) \mid\ & p \in Ports \wedge !p.isReplicated \wedge TR \subseteq TaskReference \\
& \wedge (\forall t, t' \in TR : p \in t.actor.Writes \wedge t.stop = t'.stop) \\
& \wedge \neg(\exists t' \in TaskReference \setminus TR, \exists t \in TR : t.stop = t'.stop \\
& \quad \wedge p \in t'.actor.Writes)\} \\
\cup \{(p, TR) \mid\ & p \in Ports \wedge p.isReplicated \wedge TR \subseteq TaskReference \\
& \wedge (\forall t, t' \in TR : p \in t.actor.Writes \wedge t.stop = t'.stop \wedge t.cpu = t'.cpu) \\
& \wedge \neg(\exists t' \in TaskReference \setminus TR, \exists t \in TR : t.stop = t'.stop \\
& \quad \wedge t.ecu = t'.ecu \wedge p \in t'.actor.Writes)\}
\end{aligned} \tag{4.7}$$

Calculate Local Schedules

The result of the previous steps is a flattened schedule that considers also non-functional aspects. This global schedule must be mapped into local schedules for each ECU. Only elements that are related to an ECU are copied into the local schedule. The local schedule `schedule(e, Sched)` for an ECU `e` and a global schedule `Sched` is calculated as follows:

$$\begin{aligned}
schedule(e, Sched) = & \{t \in TaskReference \mid t \in Sched \wedge t.ecu = e\} \\
& \cup \{i \in InputReference \mid i \in Sched \wedge i.ecu = e\} \\
& \cup \{o \in OutputReference \mid o \in Sched \wedge o.ecu = e\} \\
& \cup \{tr \in TriggerReference \mid tr \in Sched \wedge e \in tr.ECU\} \\
& \cup \{cr \in CommunicationReference \mid cr \in Sched \wedge \\
& \quad \neg cr.p.isReplicated \wedge (cr.ar.actor.ecu = e \vee e \in cr.p.ReadECUs)\} \\
& \cup \{pub \in Publish \mid pub \in Sched \wedge e \in pub.p.ReadECUs \wedge \\
& \quad \neg cr.p.isReplicated \vee (cr.AR.get(0).actor.ecu = e)\}
\end{aligned} \tag{4.8}$$

Sort Schedule

The last step sorts the local schedule, a set of schedule references, according to the execution time. The order of elements at the same logical moment depends on the value

of the microTime attribute. The only peculiarity is the handling of the communication references. Since the communication medium is not infinitely fast and also temporal synchronization imprecision has to be considered, the receiver has to wait for up to the maximal latency plus the maximal synchronization imprecision until the message is received. Therefore, the local schedule executes first all send operations and then starts with the receive operations to minimize the required time.

4.4.5 Operational Semantics

This section presents a precise definition of the operational semantics. For reasons of simplicity, the explanation is restricted to a system consisting of only one job. Since race conditions are excluded by design, further jobs can be executed logically in parallel. The actual sequence of actions can be derived by merging the parallel sequences. The fault-tolerance mechanisms are not considered in this section. If a fault is detected, the system is simply aborted. The actual handling of faults is explained in the context of the fault-tolerance model.

Program Configuration

Analog to Giotto, a program configuration $\mathtt{C_j}, \mathtt{e} = (\mathtt{m}, \delta, \mu\delta, \mathtt{v}, \mathtt{trigger}, \tau)$ for a job $\mathtt{j}$ on an ECU $\mathtt{e} \in \mathtt{ECU}$ consists of a mode $\mathtt{m} \in \mathtt{j.Modes}$, the current logical moment $\delta \in \mathbb{Q}$, the current micro time $\mu\delta \in \mathbb{N}$, a valuation of the ports $\mathtt{v}$ for the set of relevant global and private ports, the decisions of all triggers $\mathtt{trigger} : \mathtt{Trigger} \rightarrow \mathbb{N}$ and a time stamp $\tau \in \mathbb{Q}$. The relevant local schedule for mode $\mathtt{m}$ on ECU $\mathtt{e}$ is defined by $\mathtt{Sched_{e,m}}$.

The set of relevant ports is determined by the global ports that are read by actors or triggers on the ECU $\mathtt{P_e}$ and the set of relevant **private ports** $\mathtt{P_{priv,e}} = \mathtt{P_{local,e}} \cup \mathtt{P_{comm,e}}$. The private ports $\mathtt{P_t} = \mathtt{P_{t,read}} \cup \mathtt{P_{t,write}}$ for a task $\mathtt{t}$ reflect the variables in the task stub used for the copy operations at the logical start and end of the task execution. Furthermore, also private ports $\mathtt{P_i}$ for each input $\mathtt{i}$ are required to store the tentative input results. The set of relevant local private ports on an ECU $\mathtt{e}$ is $\mathtt{P_{local,e}} = \bigcup_{\mathtt{t} \in \mathtt{Tasks_e}} \mathtt{P_t} \cup \bigcup_{\mathtt{i} \in \mathtt{Input_e}} \mathtt{P_i}$. In addition, the relevant results of actors executed on other ECUs are needed as well. These results are received by the communication interface and stored in $\mathtt{P_{comm,e}} = \{\mathtt{cr.p} \mid \mathtt{cr} \in \mathtt{CommunicationReferences} \wedge \mathtt{cr} \in \mathtt{Sched_{e,m}} \wedge \mathtt{cr.ar.actor.ecu} \neq \mathtt{e}\}$. Private ports have the same attributes than global ports, but in addition an attribute referring to the actor identifier.

Micro Steps

Analog to Giotto's **micro steps**, micro steps are defined for FTOS. A micro step at the logical moment δ executes all actions scheduled at time `delta` with a specific micro time $\mu\delta$. In contrast to Giotto, each of the micro steps can be executed independently of the other micro steps. This gives a maximum flexibility regarding program changes: the system can jump to an arbitrary program counter $(\delta, \mu\delta)$ as reaction towards the detection of an error. The initial program configuration is defined by $(\mathtt{j.start}, 0, 0, \mathtt{Init(v)}, *, 0)$. Before starting the program execution, each ECU tries to

integrate into the potentially running system and copy the program configuration from other units. Only if the system is currently not running, the ECU tries to synchronize with other ready ECUs and start the system execution as soon as enough ECUs are available. The required ECUs are defined in the fault model.

For a given program configuration $\mathtt{C_{j,e}} = (\mathtt{m}, \delta, \mu\delta, \mathtt{v}, \mathtt{trigger}, \tau)$, the successor configuration $\mathtt{C'_j}, \mathtt{e}$ results from one of the following steps:

1. [**Stop Task**, $\mu\delta = 0$]: Let $\mathtt{Stop} = \{\mathtt{t} \in \mathtt{Task} \mid \mathtt{tr} \in \mathtt{TaskReference} \wedge \mathtt{tr} \in \mathtt{Sched_{e,m}} \wedge \mathtt{tr.stop} = \delta \wedge (\forall(\mathtt{tr}, \mathtt{cond}) \in \mathtt{tr.TriggerList} : \mathtt{cond} = \mathtt{trigger(tr)})\}$ be the set of tasks that complete their execution at the logical moment δ. For each task $\mathtt{t} \in \mathtt{Stop}$, the run-time system checks whether the physical execution of $\mathtt{t}$ was successfully completed. Let $\mu\delta' = -1$ if a deadline violation is detected by the scheduler (to abort the execution) or $\mu\delta' = 1$ otherwise.

   For all completed tasks $\mathtt{t} \in \mathtt{Stop}$, the result of the task execution is available in the private ports $\mathtt{P_t}$. Let

$$v'(p) = \begin{cases} t.function(v(P_{t,read}))(p) & , \exists t \in Stop : p \in P_{t,write} \\ v(p) & , otherwise \end{cases} \tag{4.9}$$

   be the valuation function that agrees with the results of the completed tasks in their private write ports and otherwise with $\mathtt{v(t)}$. The successor configuration is obtained by $\mathtt{C'_{j,e}} = (\mathtt{m}, \delta, \mu\delta', \mathtt{v'}, \mathtt{trigger}, \tau)$.

2. [**1st Send Phase**, $\mu\delta = 1$]: Let $\mathtt{Send} = \{\mathtt{cr} \in \mathtt{CommunicationReference} \mid \mathtt{cr} \in \mathtt{Sched_{e,m}} \wedge \mathtt{cr.ar.actor.ecu} = \mathtt{e} \wedge \mathtt{cr.ar} \in \mathtt{TaskReference} \wedge \mathtt{cr.ar.stop} = \delta \wedge (\forall(\mathtt{tr}, \mathtt{cond}) \in \mathtt{cr.ar.TriggerList} : \mathtt{cond} = \mathtt{trigger(tr)})\}$ be the set of messages that should be sent at this program point. The set of messages is handed to the networking component for further processing. Whether the communication is actually executed depends on the networking component that might perform optimizations, for instance by skipping a message, if the value of the port did not change. Consensus protocols are performed by the networking component, if required. The successor configuration is obtained by $\mathtt{C'_{j,e}} = (\mathtt{m}, \delta, 2, \mathtt{v}, \mathtt{trigger}, \tau)$.

3. [**1st Receive Phase**, $\mu\delta = 2$]: Let $\mathtt{Receive} = \{\mathtt{cr} \in \mathtt{CommunicationReference} \mid \mathtt{cr.ar} \in \mathtt{TaskReference} \wedge \mathtt{cr} \in \mathtt{Sched_{e,m}} \wedge \mathtt{cr.ar.actor.ecu!} = \mathtt{e} \wedge \mathtt{cr.ar.stop} = \delta\}$ be the set of messages that should be received at this program point. Let $\mathtt{val(cr.p)}$be the received value. If no message is received, let $\mathtt{val(cr.p)} = \mathtt{v(p)}$. Let

$$v'(p) = \begin{cases} val(p) & , \exists cr \in Receive : p = cr.p \\ v(p) & , otherwise \end{cases} \tag{4.10}$$

   be the valuation function that agrees with the received values for all private ports $\mathtt{Ports_{cr.ar,cr.p}} \subseteq \{\mathtt{P_{priv\_comm,e}} \mid \mathtt{cr} \in \mathtt{recMsgs}\}$ and with $\mathtt{v}$ otherwise. The successor configuration for this step is $\mathtt{C'_{j,e}} = (\mathtt{m}, \delta, 3, \mathtt{v'}, \mathtt{trigger}, \tau)$.

4. [**Task Publish Phase**, $\mu\delta = 3$]: Let $\mathtt{Pub} = \{\mathtt{pub} \in \mathtt{Publish} \mid \mathtt{pub.ar.get(0)} \in \mathtt{TaskReference} \wedge \mathtt{pub.ar.get(0).stop} = \delta\}$ be the set of publish objects that refer to task stop events at time δ. For each publish object $\mathtt{pub} \in \mathtt{Pub}$, let $\mathtt{P_{pub}} = \{\mathtt{p} \in \mathtt{P_{priv,e}} \mid \mathtt{pub.p.p} = \mathtt{p.p} \wedge \exists \mathtt{a} \in \mathtt{pub.AR.actor} : \mathtt{p} \in \mathtt{a.writes}\}$ be the set

of relevant tentative results. Let

$$v'(p) = \begin{cases} pub.p.unify(P_{pub}) & , \exists pub \in Pub : p = pub.p \\ v(p) & , otherwise \end{cases} \tag{4.11}$$

the valuation function that considers the results of the publish operations for all port associated with the publish operations in Pub and with v otherwise. For ports without unifying strategy, the identity function is added as unifying strategy. Each unifying strategy is designed to handle flexible numbers of result to cope with fault effects. Results from actors that are assumed to be erroneous are not considered. If no correct result for one publish object is available, the execution is aborted by setting $\mu\delta' = -1$, otherwise $\mu\delta'$ is set to 4. The successor configuration C' for the fourth step is $C'_{j,e} = (m, \delta, \mu\delta', v', trigger, \tau)$.

5. [**Output Trigger Phase**, $\mu\delta = 4$]: The triggers that were not evaluated during the current model cycle and whose results influence the execution of an output scheduled at the logical time δ are evaluated during this step. The set of triggers is defined by $OutTrigger = \{t \in Trigger | \exists tr \in TriggerReference \wedge tr.trigger = t \wedge tr \in Sched_{e,m} \wedge tr.start = \delta \wedge tr.microTime = BeforeOutput\}$. Let

$$trigger'(t) = \begin{cases} trigger.function(v(t.Reads))(t) & , t \in OutTrigger \\ trigger(t) & , otherwise \end{cases} \tag{4.12}$$

be the valuation function that refers to the trigger results. The function is generated during the M2M transformation based on the information contained in the model. Let

$$v'(p) = \begin{cases} trigger.function(v(t.Reads))(p) & , \exists t \in OutTrigger : p \in trigger.Writes \\ v(p) & , otherwise \end{cases} \tag{4.13}$$

be the valuation function that updates the ports with the results of the trigger execution. The successor state for the output trigger phase is defined by $C'_{j,e} = (m, \delta, 5, v', trigger', \tau)$.

6. [**Output Phase**, $\mu\delta = 5$]: Let $Enabled = \{o \in Output | \exists or \in OutputReference : o = or.output \wedge or \in Sched_{e,m} \wedge or.start = \delta \wedge (\forall (tr, cond) \in or.TriggerList : cond = trigger(tr))\}$ be the set of enabled outputs that should be executed at this logical moment. The execution of the output actors in Enabled is based on the values of the relevant port, but has no influence on the program execution, because only environment variables are changed. The successor configuration is defined by $C'_{j,e} = (m, \delta, 6, v, trigger, \tau)$.
7. [**Input Trigger Phase**, $\mu\delta = 6$]: Analog to the output trigger phase, the triggers that were not evaluated during the current model cycle and whose results influence the execution of an input scheduled at the logical time δ are evaluated during this step. The set of triggers is defined by $InputTrigger = \{t \in Trigger | \exists tr \in TriggerReference \wedge tr.trigger = t \wedge tr \in Sched_{e,m} \wedge tr.start = \delta \wedge tr.microTime = BeforeInput\}$. Let trigger'(t) and v'(p) be functions that are defined analog to the functions in the output

trigger phase. The successor state for the input trigger phase is defined by $\mathtt{C}'_{\mathtt{j},\mathtt{e}} = (\mathtt{m}, \delta, 7, \mathtt{v}', \mathtt{trigger}', \tau)$.

8. [**Input Phase**, $\mu\delta = 7$]: Let $\mathtt{Enabled} = \{\mathtt{i} \in \mathtt{Input} | \exists\ \mathtt{ir} \in \mathtt{InputReference} : \mathtt{i} = \mathtt{ir.input} \wedge \mathtt{ir} \in \mathtt{Sched}_{\mathtt{e},\mathtt{m}} \wedge \mathtt{ir.start} = \delta \wedge \forall(\mathtt{tr}, \mathtt{cond}) \in \mathtt{ir.TriggerList} : \mathtt{cond} = \mathtt{trigger}(\mathtt{tr})\}$ be the set of enabled inputs that should be executed at this logical moment. Let

$$v'(p) = \begin{cases} i.function()(p) & ,\exists i \in Enabled \wedge p \in P_i \\ v(p) & ,otherwise \end{cases} \tag{4.14}$$

be the valuation function that agrees with the new results of the inputs in the relevant private ports and with v otherwise. The successor configuration is obtained by $\mathtt{C}'_{\mathtt{j},\mathtt{e}} = (\mathtt{m}, \delta, 8, \mathtt{v}', \mathtt{trigger}, \tau)$.

9. [**2nd Send Phase**, $\mu\delta = 8$]: The second send phase is performed analog to the first send phase. Let $\mathtt{Send} = \{\mathtt{cr} \in \mathtt{CommunicationReference} | \mathtt{cr} \in \mathtt{Sched}_{\mathtt{e},\mathtt{m}} \wedge \mathtt{cr.ar.actor.ecu} = \mathtt{e} \wedge \mathtt{cr.ar} \in \mathtt{InputReference} \wedge \mathtt{cr.ar.start} = \delta \wedge (\forall(\mathtt{tr}, \mathtt{cond}) \in \mathtt{cr.ar.TriggerList} : \mathtt{cond} = \mathtt{trigger}(\mathtt{tr}))\}$ be the set of messages that should be sent at this program point. The successor configuration is obtained by $\mathtt{C}'_{\mathtt{j},\mathtt{e}} = (\mathtt{m}, \delta, 9, \mathtt{v}, \mathtt{trigger}, \tau)$.
10. [**2nd Receive Phase**, $\mu\delta = 9$]: Let $\mathtt{Receive} = \{\mathtt{cr} \in \mathtt{CommunicationReference} | \mathtt{cr.ar} \in \mathtt{InputReference} \wedge \mathtt{cr} \in \mathtt{Sched}_{\mathtt{e},\mathtt{m}} \wedge \mathtt{cr.ar.actor.ecu} \neq \mathtt{e} \wedge \mathtt{cr.ar.start} = \delta\}$ be the set of messages that should be received at this program point. Let $\mathtt{val}(\mathtt{cr.p})$be the received value. If no message is received, let $\mathtt{val}(\mathtt{cr.p}) = \mathtt{v}(\mathtt{p})$. The new valuation function $\mathtt{v}'$ can be calculated analog to the first receive phase. The successor configuration for this step is $\mathtt{C}'_{\mathtt{j},\mathtt{e}} = (\mathtt{m}, \delta, 10, \mathtt{v}', \mathtt{trigger}, \tau)$.
11. [**Input Publish Phase**, $\mu\delta = 10$]: Let $\mathtt{Pub} = \{\mathtt{pub} \in \mathtt{Publish} \mid \mathtt{pub.ar.get}(0) \in \mathtt{InputReference} \wedge \mathtt{pub.ar.get}(0).\mathtt{start} = \delta\}$ be the set of publish objects that refer to input operations at the logical moment. Let $\mathtt{v}'$ be the new valuation function that is defined analog to the task publish phase and considers the publish objects $\mathtt{pub} \in \mathtt{Pub}$. Let $\mu\delta' = -1$ if for at least publish object, no valid results is available and $\mu\delta' = 11$ otherwise. The successor state is obtained by $\mathtt{C}'_{\mathtt{j},\mathtt{e}} = (\mathtt{m}, \delta, \mu\delta', \mathtt{v}', \mathtt{trigger}, \tau)$.
12. [**Task Trigger Phase**, $\mu\delta = 11$]: Analog to the other trigger phases, the triggers that were not evaluated during the current model cycle and whose results influence the execution of a task start scheduled for the logical moment are evaluated during this step. The set of triggers is defined by $\mathtt{TaskTrigger} = \{\mathtt{t} \in \mathtt{Trigger} | \exists \mathtt{tr} \in \mathtt{TriggerReference} \wedge \mathtt{tr.trigger} = \mathtt{t} \wedge \mathtt{tr} \in \mathtt{Sched}_{\mathtt{e},\mathtt{m}} \wedge \mathtt{tr.time} = \delta \wedge \mathtt{tr.microTime} = \mathtt{BeforeTask}\}$. Let trigger′(t) and v′(p) be functions that are defined analog to the functions in the other trigger phases. The successor state for the task trigger phase is defined by $\mathtt{C}'_{\mathtt{j},\mathtt{e}} = (\mathtt{m}, \delta, 12, \mathtt{v}', \mathtt{trigger}', \tau)$.
13. [**Start Task**, $\mu\delta = 12$]: Let $\mathtt{Start} = \{\mathtt{t} \in \mathtt{Task} \mid \mathtt{tr} \in \mathtt{TaskReference} \wedge \mathtt{tr} \in \mathtt{Sched}_{\mathtt{e},\mathtt{m}} \wedge \mathtt{tr.start} = \delta \wedge (\forall(\mathtt{tr}, \mathtt{cond}) \in \mathtt{tr.TriggerList} : \mathtt{cond} = \mathtt{trigger}(\mathtt{tr}))\}$

be the set of tasks that need to be started at this logical moment. Let

$$v'(p) = \begin{cases} v(p_g) & ,\exists t \in Start, \exists p_g \in P_e : p \in P_{t,read} \wedge p_g.p = p.p \\ v(p) & ,otherwise \end{cases} \tag{4.15}$$

be the valuation function that realizes the copy operation of the values of the global ports into the input ports of the tasks that are started logically. The physical execution of the task function is realized by the scheduler. The task set `Start` is send to the scheduler, which is responsible for the physical execution. The successor state for the task start phase is defined by $\mathtt{C}'_{\mathtt{j,e}} = (\mathtt{m}, \delta, 13, \mathtt{v}', \mathtt{trigger}, \tau)$.

14. [**Wait For Next Logical Moment**, $\mu\delta = 13$]: The last action of the system at a logical moment is to wait for the next point in time, when the system should be executed. This point in time can be derived by finding the earliest next element $\mathtt{sr} \in \mathtt{Sched_{e,m}} : (\mathtt{sr.start} > \delta \vee \mathtt{sr.stop} > \delta) \wedge \neg(\exists \mathtt{sr}' \in \mathtt{Sched_{e,m}}$: $(\mathtt{sr.start} > \delta \vee \mathtt{sr.stop} > \delta) \wedge \mathtt{pos}(\mathtt{Sched_{e,m}}, \mathtt{sr}') < \mathtt{pos}(\mathtt{Sched_{e,m}}, \mathtt{sr}'))$. If no next element is available ($\mathtt{sr} = \perp$), the system waits for the end of the mode cycle or for the end of the idle time. Let

$$\delta' = \begin{cases} sr.start - \delta & ,sr \neq \perp sr.start > \delta \\ sr.stop - \delta & ,sr \neq \perp sr.start \leq \delta \\ m.duration - \delta & ,sr = \perp \wedge m \neq \perp \\ j.idleTime & ,otherwise \end{cases} \tag{4.16}$$

The system execution is suspended for the time interval $\varepsilon = \delta' - \delta$. Let $\mu\delta' = 0$ if $\mathtt{sr} \neq \perp$, and $\mu\delta = 14$ otherwise. The successor state is defined by $\mathtt{C}'_{\mathtt{j,e}} = (\mathtt{m}, \delta', \mu\delta, \mathtt{v}, \mathtt{trigger}, \tau + \varepsilon)$.

15. [**Determination of Next Mode**, $\mu\delta = 14$]: The last action of the mode cycle is to determine the next mode that should be executed. Let $\mathtt{m}' = \mathtt{j.change.function}(\mathtt{m}, \mathtt{v}(\mathtt{j.change.Reads}))$ be the new mode calculated by the mode change function generated during M2M transformation based on the finite state automaton specified in the software model. Let

$$v'(p) = \begin{cases} j.change.function(m, v(j.change.Reads))(p) & ,p \in j.change.Writes \\ v(p) & ,otherwise \end{cases} \tag{4.17}$$

be the valuation function that refers to the results of the mode change function for all ports written by this function and to v otherwise. Let $\mu\delta' = 0$ if the system executes a mode $\mathtt{m}' \neq \perp$ and $\mu\delta' = 13$ otherwise. The successor state of the mode determination phase is defined by $\mathtt{C}'_{\mathtt{j,e}} = (\mathtt{m}', 0, \mu\delta', \mathtt{v}', \mathtt{trigger}, \tau)$.

4.5 Fault Model

The implementation of the different fault-tolerance mechanisms depends on the probability, type and location of the possible faults. The assumptions about the fault behavior

of the individual components are contained in the fault hypothesis, which is a result of the system's analysis phase. The fault hypothesis states the types and numbers of faults that have to be tolerated by the system. It is therefore the most important document in the design of fault-tolerant systems [Kop06].

The information contained in the fault hypothesis can be used in three ways: to define appropriate fault-tolerance mechanisms, to select concrete components realizing the run-time system, and to verify the concrete implementation of the generated run-time system with respect to the assumed faults. In general, the fault hypothesis should contain the following information:

- Location of potential faults: The fault hypothesis must state which components can be affected by a fault.
- Effect of a fault: The consequences of the activation of a fault have to be stated. The effects range from fail-silent behavior such as a lost message to arbitrary effects such as bit corruption in memory.
- Type of potential faults: Besides the effect, also the failure mode must be specified. Possible modes are permanent, soft-permanent or transient failures.
- Probability of the faults: Information about the probability of a fault activation is necessary to assess the reliability / safety of a system.
- Assumptions: It is not useful to design the system for all possible fault configurations, because some might be very improbable. The fault hypothesis must therefore state basic assumptions such as the maximal number of faults that need to be handled in parallel.

State of the Art

A number of techniques and tools are available to support the developer during the safety analysis phase. Examples for techniques are the fault tree analysis [VGRH81], the Hazard and Operability Analysis (HAZOP) [MNP95], and the failure mode and effect analysis (FMEA) [Rei79]. These techniques help in identifying the initial faults / detecting hazards and are based on (semi-)formal methods. However, the resulting fault hypothesis is usually documented only in textual form. Very often the information is spread over different documents and important assumptions made during implementation are not documented or only within the code. Furthermore, the fault hypothesis is often mixed with fault-tolerance mechanisms.

Figure 4.15 gives an example for the fault hypothesis of TTA [Kop06]. As the fault hypothesis is designed for applications using TTA, it is designed in a generic way. One typical problem of textual fault hypothesis is redundancy. Assumption 3, for example, is further refined in assumptions 6 and 7 without referring to each other. Another issue is the spreading of information: the assumption that each node forms a FCU is specified in the first assumption. Further assumptions on the failure behavior of a node can be found in assumptions 3, 5, 6 and 7. Another problem is ambiguity: by the term *arbitrary failure* different developers might think of different fault effects. Furthermore, the hypothesis contains already implementation details. Assumptions 8 to 12 are no

1. *A node computer forms a single FCR.*
2. *A communication channel including the central guardian forms a single FCR.*
3. *A node computer can fail in an arbitrary failure mode.*
4. *A central guardian distributes the message received from the node computers. It can fail to distribute the messages, but cannot generate messages on its own.*
5. *The permanent failure rate of a node or the central guardian is in the order of 100 FIT[a] {...}*
6. *The transient failure rate of a node is in the order of 100.000 FIT {...}*
7. *One out of about fifty failures of a node computer is non-fail silent.*
8. *The central guardian transforms the non-fail silent and the slightly-out-of-specification (SOS) failures of the node computers in the temporal domain to fail-silent failures in the temporal domain {...}*
9. *The detection of a single error is performed by a membership algorithm. The error detection latency is less than two TDMA rounds.*
10. *The detection of multiple errors is performed by a clique avoidance algorithm. The detection latency is less than two TDMA rounds.*
11. *The system can recover from a single transient fault within two TDMA rounds.*
12. *The system can recover from a massive transient that destroys the clock synchronization within 8 TDMA rounds after the transient has disappeared.*
13. *The state repair time of an application takes an application specific amount of time which must be derived from knowledge about the application software.*

[a] Failures in Time: the number of failures in 10^9 hours

Figure 4.15: Example of a Textual Fault Hypothesis [Kop06]

assumptions on faults, but statements about the implementation of the fault-tolerance mechanisms.

It is obvious that in the context of FTOS a formal approach is required. However, most available approaches use application logic for the formal specification which contradicts the goal of generality of FTOS. Arora et al. [AG93], for instance, add fault transitions directly at application level. In contrast, the goal of FTOS is to describe the fault hypothesis in a formal, unambiguous way based on an application independent fault effects.

4.5.1 Main Concepts

The main concepts of the fault model are to rely on generic fault effects instead of application-specific faults and to force the developer to state all fault assumptions explicitly. The model consists of

- A definition of the fault containment units (FCU),

- The effect of the assumed faults on components of the FCU,
- The fault configurations that need to be considered, and
- The assumptions on the activation of faults.

Fault Containment Unit (FCU)

A fault containment unit (FCU) [HTBSL78] defines the components (the region) that may be affected by a set of faults. Within the FCU, the faults must be handled by fault-tolerance mechanisms to ensure that resulting erroneous values do not propagate beyond the boundaries of the FCU.

Some authors distinguish between fault and error containment and accordingly between fault and error containment regions/units [BS05]. According to the distinction, the first term refers to the components that are directly affected by a fault, while the second term refers to the components that are affected by the resulting error. Fault containment can be achieved by physical separation of the individual fault containment units to exclude common cause failures. Error containment is done by additional mechanisms, e.g. tests or voters, implemented in software or hardware. The thesis uses the term fault containment unit in the classical sense [HTBSL78, Kop06] as all components that might be affected by faults and their resulting errors.

The abstract syntax of a FCU is $(\texttt{f}, \texttt{Components}, \texttt{Behavior}, \texttt{Children}, \texttt{silence}) \in \texttt{FCU}$. The definition describes the set of components `Components` that form the FCU `f`. `Components` can comprise both hardware and software components that were defined in the hardware and software model. To simplify the specification, only the highest-level component must be specified, all sub components are automatically included. A FCU consisting of an ECU `e` comprises for example all hardware components of the ECU and all actors executed on `e`. The fault effects are specified at component level using the $\texttt{Behavior} \subseteq \texttt{ComponentBehavior}$ reference. The abstract syntax to describe the behavior of a set of components is $(\texttt{C}, \texttt{F}) \in \texttt{ComponentBehavior}$. It describes the set of components $\texttt{C} \subseteq \texttt{Components}$ that may be affected by the faults specified in $\texttt{F} \subseteq \texttt{Fault}$.

In contrast to other approaches [Kop06] that require fault containment units to be distinct, FTOS allows a hierarchical arrangement of FCUs. This is necessary because the fault-tolerance mechanisms are triggered in FTOS by changes of the state of the FCUs. Therefore, a hierarchical arrangement allows a more sophisticated error reaction. The activation of a specific fault, e.g. a wrong sensor result, can be tolerated directly, e.g. by taking the previous measurement result, while the activation of an arbitrary fault of one of the components of the ECU can lead to the exclusion of this module in a TMR system.

In principle, the fault containment requirement demands that the behavior at the boundaries is consistent with the specified behavior. However, it might not be possible to maintain this behavior in the presence of erroneous components. A typical approach is to switch into fail-silence mode, meaning that all interactions observed at the FCU boundary are correct, but that some interactions might be inhibited due to detected errors. However, the implementation of the adjacent regions depends on the failure be-

havior of the FCU. A network can for example be designed to form a FCU that provides a reliable communication. This assumption simplifies the implementation of consensus protocols. If on the other hand, the FCU does not provide reliable communication, but may also behave fail-silent, the consensus protocol design gets more complex. The information, whether a FCU is assumed to be reliable or if also fail-silent behavior has to be considered, is specified in the attribute `silence`.

Fault and Fault Effects

Most approaches describe a fault by the application-specific causes. With knowledge about the application, the effect can be derived. A broken communication link or a short-circuit are causes that lead to the loss of a message. Radiation may lead to message corruption. It is obvious that there are a number of application specific causes. Hence, the approach on describing the causes is not useful for a generic approach. Instead, the idea of FTOS is to rely on application-independent fault effects. The abstract syntax of the fault class is $(\texttt{effect}, \texttt{extent}, \texttt{mode}, \texttt{duration}, \texttt{probability}) \in \texttt{Fault}$. The main attribute is the definition of the effect. FTOS provides for each component type a list of possible, generic fault effects. For network components for example, FTOS defines seven different fault effects as suggested in the international standard IEC 61508 [Int98]: `DataCorruption`, `TimeDelay`, `DeletedTelegram`, `Repetition`, `InsertedTelegram`, `ResequencedTelegram`, `AddressingError`, `Masquerade`. Also combinations of different effects are possible: a fault with an arbitrary effect for example combines all possible fault effects of the contained components and can be specified with `Arbitrary` conveniently. This solves also one previously mentioned problem: the arbitrary behavior is unambiguously specified using this concept.

Another important property of a fault is the `extent` of the effect. The valid values of `extent` depend on the specific effect. For example, one can describe the amount of bits that may be corrupted, if $\texttt{effect} = \texttt{DataCorruption}$. Furthermore, the developer can specify the fault mode `mode`. Possible values are `Permanent`, `SoftPermanent`, or `Transient`. The difference between a permanent and a soft permanent is the persistence of a failure. In the context of memory, a permanent `DataCorruption` would be a bit that is always high or low. A soft permanent `DataCorruption` would be a bit-flip that becomes obsolete the next time the bit is written. For transient faults, the developer can specify the length of the activation period. The probability of a fault `probability` can be used to assess whether the reliability / safety of a system design meets the requirements.

Fault Configurations Sets

Besides the definition of the different faults and their effects, information about the number of faults that need to be tolerated simultaneously is important. This information is the major foundation for the selection of appropriate fault-tolerance mechanisms and the necessary degree of redundancy.

FTOS uses fault configuration sets to describe the different fault states of a system. A **fault configuration** specifies the set of correct and the set of erroneous FCUs. The re-

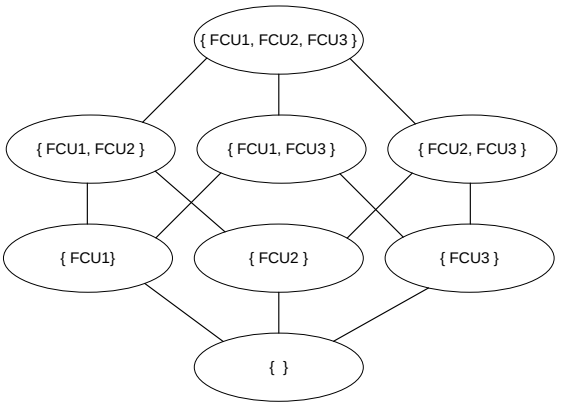


Figure 4.16: Haase Diagram of Possible Fault Configurations

lationship between the different fault configurations can be denoted by a lattice. The configuration consisting of only correct FCUs is the top element; the configuration consisting of only faulty FCUs the bottom element of this lattice. Figure 4.16 depicts a Haase diagram for a system consisting of three units `FCU1`, `FCU2`, and `FCU3`. The sets in the figure denote the correct FCUs. The activation of a fault leads to a fault configuration in a level below the current fault configuration. The successful recovery from a fault leads to a fault configuration in the level above the current fault configuration.

Fault configurations play an important role, as they may represent different operational modes. An example is a triple modular redundancy system using semi-active replication, where each replica forms one fault containment unit. The possible fault configurations are depicted in Figure 4.16. If no implementation details about the master selection process are known, each fault configuration represents one operational mode. The upper level consists of the fault configuration $\{\texttt{FCU1}, \texttt{FCU2}, \texttt{FCU3}\}$, where all replicas can serve as master and the system is executed reliable and safe forms exactly one operational mode. At the next level, the master can be selected out of the correct replicas and the system is executed in safe mode. Here three different operational modes are available due to the three sets of correct nodes. The same is true for the third level, where the correct unit switches into fail-safe state.

However, some fault configurations may be so unlikely that they are not considered during system design. In FTOS, the developer has to specify all fault configurations that should be considered. Because the number of fault configurations grows exponentially with the number of FCUs, FTOS allows to define sets of fault configurations. Each set represents a relevant system mode. A fault configuration set is described by the abstract syntax $(\texttt{fc}, \texttt{Correct}, \texttt{Erroneous}, \texttt{Implicit}, \texttt{start}) \in \texttt{FaultConfigurationSet}$. It can be interpreted as a function that maps a subset $\texttt{Correct} \cup \texttt{Erroneous} \subseteq \texttt{FCU}$ of fault containment units to a Boolean value. If a fault configuration is active, all units in `Correct` are assumed to be correct, while all units in `Erroneous` are erroneous.

Several fault configurations can be combined to form one set. The first possibility is to not specify the fault state of a certain FCU. In addition, the developer can specify so-called implicit configurations by $\texttt{Implicit} \subseteq \texttt{ImplicitConfiguration}$. An im-

plicit configuration $(\texttt{FCUs}, \texttt{min}, \texttt{max}) \in \texttt{ImplicitConfiguration}$ describes a set of FCUs $\texttt{FCUs} \subseteq \texttt{FCU}$ and the assumed minimal $\texttt{min} \in \mathbb{N}$ and maximal $\texttt{max} \in \mathbb{N}$ number of faulty units at a time. It is important that no fault configuration is contained in two distinct fault configuration sets. This is validated by FTOS during the model validation phase. Finally, the Boolean value `start` determines whether the system is allowed to start the execution, if the system assumes the relevant fault configuration set to be active.

Fault Activation Assumption

Besides the assumptions on the number of parallel faults, also assumptions on the temporal activation of faults are very important. These assumptions are typically made during the implementation phase and not documented thoroughly. A common example is the assumption that during the reaction of a system to the activation of one fault, no other fault is activated. This assumption contains of course much knowledge about the real implementation: the detection latency and the duration of the reaction. Since the fault hypothesis should be independent of the concrete implementation, this approach is not reasonable. Even worse, it is not possible to have this knowledge in component-based approaches, since the concrete implementation is not known.

In FTOS, the developer is forced to state these assumptions explicitly. Analog to mean-time between failure (MTBF) assumptions, FTOS requires the developer to state the least-time between failure (LTBF) $(\texttt{FCUSet}, \texttt{duration}) \in \texttt{LTBF}$ assumptions with $\texttt{fcuSet} \subseteq \texttt{FCU}$. A LTBF object defines the minimal time `duration` that passes between the activation of faults in the set of fault containment units `FCUSet`. Based on these assumptions, FTOS can automatically derive the allowed maximal time for failure recovery and other information necessary for implementation.

4.5.2 Fault Hypothesis Examples

This section discusses the fault hypothesis of the demonstrator, but starts with the discussion of the previous example for a fault hypothesis in Figure 4.15. By the concept of FCUs, FTOS forces the developer to specify all faults related with components of one FCU in the context of this FCU object. This avoids the problem of distributed assumptions, redundancy and further refinements for one component. Using the concepts of fault effects and the class ComponentBehavior, the developer can specify the assumptions in a detailed way. As some information is missing or ambiguous, the formalization of the example is complicated. One example is the assumption 7. Here it is not clear, whether this assumption refers to the assumption 5 and 6, or only to 6. The initial assumptions made to validate stated fault-tolerance mechanisms specified in assumptions 8-12 are not contained in the fault hypothesis. Therefore, it is not possible to formalize the underlying assumptions. The formalized version is specified in a generic way analog to the initial fault hypothesis. It is depicted including the textual interpretation in Figure 4.17. The major benefits are the centralization of the different assumption, the unambiguous specification, and the concrete definition of the fault effects. The latter benefit may not be obvious since both the formal and the textual hypothesis use the term **Arbitrary**. However, **Arbitrary** faults are in FTOS simply the combination of all

1. *A node computer forms a single FCR. The node computer can fail in an arbitrary failure mode. The rate of permanent arbitrary failures is 2 FIT and of permanent fail-silent failures 98 FIT. The rate of transient arbitrary failures is 2.000 FIT; the rate of transient fail-silent failures is 98.000 FIT.*
   $\forall \texttt{e} \in \texttt{ECU}:$
   $\exists \texttt{fcu} \in \texttt{FCU}: \texttt{fcu.C}_\texttt{f} = \texttt{e} \wedge$
   $\exists \texttt{cb} \in \texttt{fcu.Behavior}, \exists \texttt{f} \in \texttt{cb.F}: \texttt{f.effect} = \texttt{Arbitrary} \wedge$
   $\texttt{f.mode} = \texttt{Permanent} \wedge \texttt{f.probability} = \texttt{2FIT}$
   $\exists \texttt{cb} \in \texttt{fcu.Behavior}, \exists \texttt{f} \in \texttt{cb.F}: \texttt{f.effect} = \texttt{FailSilent} \wedge$
   $\texttt{f.mode} = \texttt{Permanent} \wedge \texttt{f.probability} = \texttt{98FIT}$
   $\exists \texttt{cb} \in \texttt{fcu.Behavior}, \exists \texttt{f} \in \texttt{cb.F}: \texttt{f.effect} = \texttt{Arbitrary} \wedge$
   $\texttt{f.mode} = \texttt{Transient} \wedge \texttt{f.probability} = \texttt{2.000FIT}$
   $\exists \texttt{cb} \in \texttt{fcu.Behavior}, \exists \texttt{f} \in \texttt{cb.F}: \texttt{f.effect} = \texttt{FailSilent} \wedge$
   $\texttt{f.mode} = \texttt{Transient} \wedge \texttt{f.probability} = \texttt{98.000FIT}$
2. *A communication channel including the central guardian* `guardian` *(modeled as a network) forms a single FCR. The guardian can be affected by faults with fail-silence effect. The rate for permanent failures is 100 FIT.*
   $\forall \texttt{n} \in \texttt{Network}:$
   $\exists \texttt{fcu} \in \texttt{FCU}: \texttt{fcu.component} = \{\texttt{n}\} \wedge$
   $\forall \texttt{g} \in \texttt{Guardian}: \texttt{contains}(\texttt{n}, \texttt{g}) \Rightarrow$
   $(\exists \texttt{cb} \in \texttt{fcu.Behavior}: \texttt{g} \in \texttt{cb.C} \wedge (\exists \texttt{f} \in \texttt{cb.F}: \texttt{f.effect} = \texttt{DeletedTelegram} \wedge$
   $\texttt{f.mode} = \texttt{Permanent} \wedge \texttt{f.probability} = \texttt{100FIT}))$

Figure 4.17: Fault Hypothesis - Formalized Version

possible fault effect types defined for the specific component. Hence, the term has an unambiguously meaning.

Fault Assumptions for Demonstrators

In the following, the assumptions for both applications are summarized. In both applications, each ECU forms a FCU. The assumed failures cover measurement errors and fail-silent behavior of the ECUs. Wrong task results are not assumed, otherwise software diversity would have to be applied. In addition to the FCUs for each ECU, a fourth FCU is specified in the elevator application to describe the failure behavior of the network. Here, transient fail-silent behavior (`DeletedTelegram`) is assumed. For the elevator application, the network is assumed to be reliable.

For the TMR application, only one failure at a time is assumed. Hence, the relevant fault configurations for the rod application are `Correct`, `FailureUnit1`, `FailureUnit2` and `FailureUnit3` and refer to the two upper levels of the Haase Diagram depicted in Figure 4.16. No assumptions are made on the status of the network FCU. However, a very important assumption is made with respect to the activation of network faults. By the use of an LTBF object, the minimum time between two lost messages is assumed to be `10ms`. The relevance of this assumption is described in Chapter 5.

The fault model of the elevator application specifies three relevant fault configuration sets:

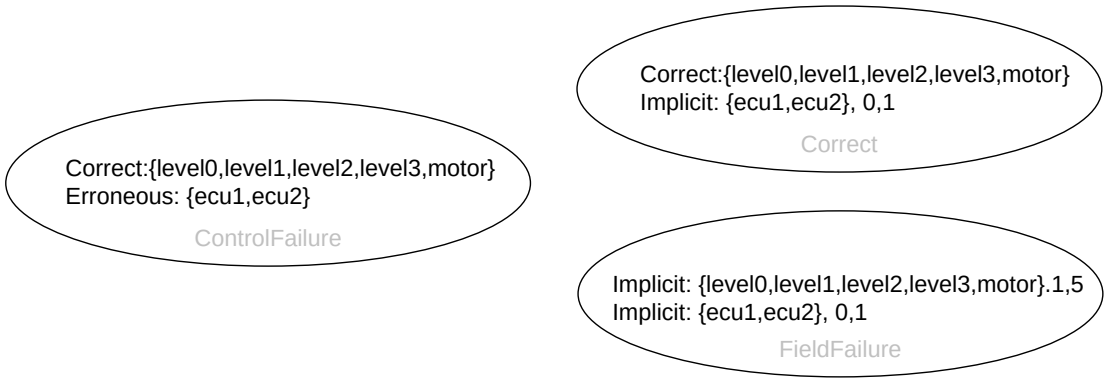


Figure 4.18: Considered Fault Configurations in the Elevator Example

- `Correct`: a configuration set with at least one correct control unit and correct field controllers,
- `FieldFailure`: a set with at least one correct control unit, but with an arbitrary non-zero number of erroneous field controllers, and
- `ControlFailure`: a set with no correct control unit, but correct field controllers.

These sets are depicted in Figure 4.18. LTBF assumptions are not specified.

4.5.3 Model-To-Model Transformation in the Context of the Fault Model

This section discusses the calculation of relevant FCUs for a fault configuration set during the M2M transformation. Section 4.6 points out that the fault-tolerance mechanisms are selected at run-time based on the currently active fault configuration set `fc` $\in$ `FaultConfigurationSet`. However, not all fault configurations are relevant for the execution on one node, so that the node only has to determine whether the system is within a relevant fault configuration set `FC` $\in$ `FaultConfigurationSet`. One obvious approach is to monitor all FCUs that `FC` refers to. This set `Rel(FC)` is derived from the equation:

$$Rel(FC) = FC.correct \cup FC.erroneous \bigcup_{i \in FC.Implicit} i.fcuSet \tag{4.18}$$

However, this set might contain more FCUs than necessary. The reason is that some fault configurations might be too unlikely to be considered and are therefore not contained in the set of fault configuration sets. The system must not take these fault configurations into account, and therefore it is sufficient to distinguish between the reachable fault configuration sets. The restriction of the number of FCUs becomes useful when optimizing the system performance by avoiding unnecessary tests.

For illustration purpose, the elevator application can be used. Figure 4.19 depicts the fault configuration of the elevator application again, but adds the direct edges of the Haase diagram between the fault configuration sets. The label of the edges specifies the FCUs that can trigger the transition between two fault configuration steps by changing

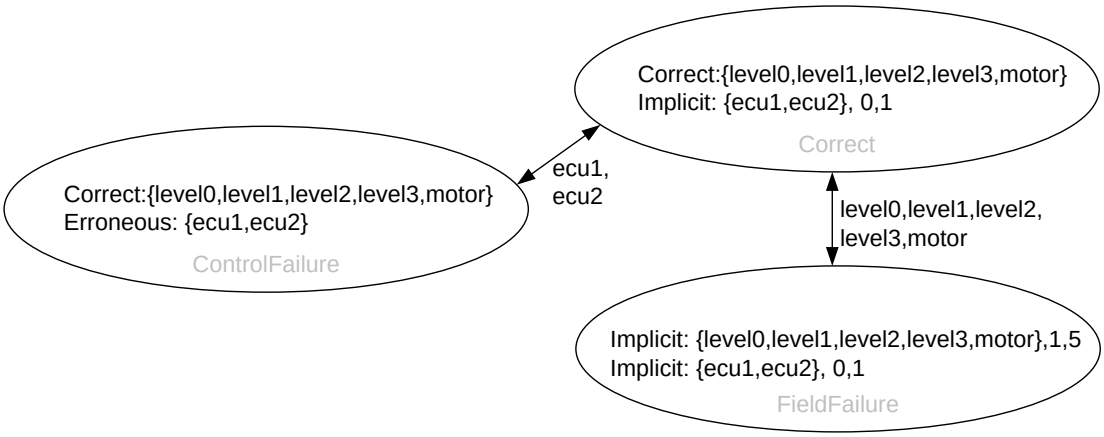


Figure 4.19: Fault Configurations and Transitions in Elevator Example

its value. The relevant set of FCUs, that is required to determine whether a fault configuration is active, is the union of all FCUs specified in the relevant edge labels. In the example, the activation of `ControlFailure` can be determined by only monitoring the status of the FCUs `ecu1` and `ecu2`. Likewise, the activation of `FieldFailure` can be determined by monitoring the status of the FCUs of the field controllers. In contrast, the status of all FCUs must be monitored to determine whether `Correct` is active.

The calculation of the edges is straightforward to implement and as a side effect also the disjunction of the different fault configuration sets can be tested. For illustration, another approach is simpler. The set of relevant fault containment units of a fault configuration set can be calculated using the set of ignored fault configurations $\texttt{FC}_{\texttt{ignored}}$, which is derived from the following formula:

$$FC_{ignored} = \bigcap_{FC \in FaultConfigurationSet} \overline{FC} \tag{4.19}$$

For optimization issues, the fault configuration set `FC` can be generalized to a set $\texttt{FC}'$. The optimal set $\texttt{FC}_{\texttt{opt}}(\texttt{FC})$ is the fault configuration set with a minimal number of referenced fault containment units. Let $|\texttt{FC.fcu}|$ denote the number of FCUs that need to be monitored.

$$\begin{aligned} FC_{opt}(FC) =& (FC \subseteq FC_{opt} \subseteq FC \cup FC_{ignored}) \wedge \\ & \forall_{FC' \neq FC_{opt}} FC \subseteq FC' \subseteq FC \cup FC_{ignored} \rightarrow |FC_{opt}.fcu| < |FC'.fcu| \end{aligned} \tag{4.20}$$

In the elevator example, the set of impossible fault configurations $\texttt{FC}_{\texttt{ignored}}$ can be cal-

culated as follows:

$$\begin{aligned} FC_{ignored} =& \overline{Correct} \wedge \overline{ControlFailure} \wedge \overline{FieldFailure} \\ =& \big(\{ecu1, ecu2\}(2,2) \vee \{field\}(1:5)\big) \\ & \wedge \big(\{ecu1, ecu2\}(0,1) \vee \{field\}(1:5)\big) \\ & \wedge \big(\{ecu1, ecu2\}(2,2) \vee \{field\}(0:0)\big) \\ =& \big(\{ecu1, ecu2\}(2,2)\big) \\ & \wedge \big(\{ecu1, ecu2\}(0,1) \vee \{field\}(1:5)\big) \\ =& \{ecu1, ecu2\}(2,2) \wedge \{field\}(1:5) \end{aligned} \tag{4.21}$$

In combination with `ControlFailure` a generalized fault configuration set is $\texttt{ControlFailure} \cup \texttt{FC}_{\texttt{ignored}} = \{\texttt{ecu1}, \texttt{ecu2}\}(2,2)$. Hence, only the status of the two control units must be monitored to determine the activation of `ControlFailure`. The relevance of this optimization is pointed out in the next section.

4.6 Fault-Tolerance Model

The fault-tolerance model specifies the employed fault-tolerance mechanisms and contains information required for correct implementation. The fault-tolerance mechanisms are specified using basic building blocks. This approach allows the reuse of the different components and the easy addition of further fault-tolerance mechanisms.

4.6.1 Main Concepts

Most of the approaches to achieve fault-tolerance mechanisms are based on the concept of detectors and correctors introduced by Kulkarni and Arora [Kul99, AK98b]. The model is very simple: detectors detect errors in the system and correctors transform the system into a correct state. This model is however not suitable for a generic approach as offline error recovery of discarded replicas is not considered by this model. Therefore, Schepers [Sch90] introduced a concept consisting of three phases: **detection**, **treatment** and **recovery**. Treatment comprehends all mechanisms that are performed online to retain a safe and/or reliable execution of the system. A typical example is the switch to a redundant channel, if the primary channel is assumed to be faulty. In contrast, recovery describes operations that are performed offline by excluded, erroneous components. In addition, recovery describes the mechanisms to integrate a successfully recovered component into the running system. These three phases fit perfectly into the concepts of FTOS. The first two phases can be executed synchronously during system execution and real-time guarantees can be provided. Most of the mechanisms can be implemented generically with little configuration information that can be extracted out of adequate models. The third phase in contrast is typically very application dependent. Here, FTOS provides only standard mechanisms such as rebooting a faulty unit or restarting a component.

However, even this concept is not detailed enough for the goals of FTOS, as also **proactive operations** are required. Proactive operations describe additional program oper-

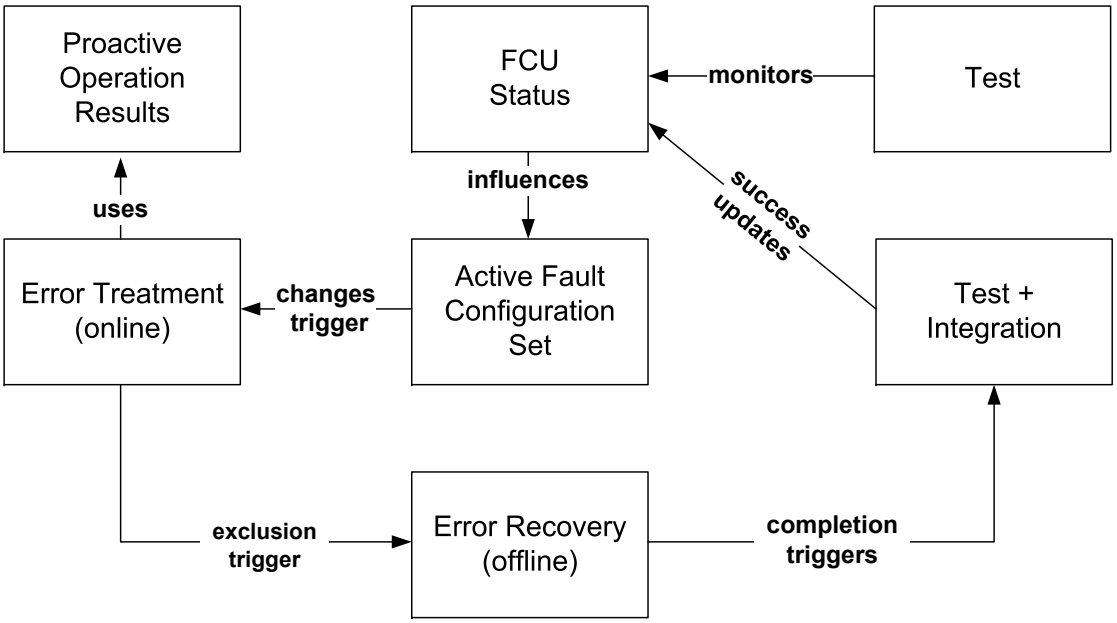


Figure 4.20: Concept of Fault-Tolerance Mechanism

ations that are not necessary for the specified application functionality, but are used to generate additional information that might be useful when an error is detected. A prominent example is the creation of checkpoints that can be used for rollback recovery. Other approaches require these operations to be executed within the application logic. However, this leads to an undesirable merge of application logic and fault-tolerance mechanisms and can be avoided by the explicit declaration of proactive operations in the fault-tolerance model.

In summary, the fault-tolerance concept in FTOS consists of four different mechanism types: proactive operations, error detection, online error treatment and offline error recovery. The relationship between the different types is illustrated in Figure 4.20. Proactive and error detection operations are integrated into the schedule and executed during the system's execution. Error detection is performed by using a set of tests. Each test monitors one or more FCUs. If at least one associated tests assumes the FCU to be faulty, the status of the FCU is set to false. Whenever the status of a FCU changes, the system determines the active fault configuration set. Changes of this set can trigger reactions by the system, the error treatment. It is performed online, but can lead to the exclusion of erroneous components. These excluded components can perform recovery operations offline followed by tests to check the correctness of the repaired component. If successful, the component can request the integration into the running system. The integration can again lead to a change of the active fault configuration set and might trigger a new round of reactions.

4.6.2 Basic Building Blocks

As discussed in the previous section, the fault-tolerance mechanisms are based on four different concepts, namely proactive operations, error detection, error treatment, and error recovery. This section presents the different components that are currently available in FTOS and discusses interesting modeling details.

Proactive Operations

Proactive operations are actions that are performed during normal system execution to generate some information that is required in case the system has to tolerate an error in the future. The most prominent example is checkpointing, the only proactive operation currently supported in FTOS. A checkpoint stores the state of a system at a specific point in time. The main advantage of the mechanism is its simplicity; the disadvantage is that only transient and soft permanent faults can be tolerated.

FTOS allows the definition of checkpoints at the level of a job. A checkpoint object $(\texttt{Scheduled}, \texttt{Ports}, \texttt{number}) \in \texttt{Checkpoint}$ describes the program counters $\texttt{Scheduled} \subseteq \texttt{ProgramCounter}$ when a checkpoint should be created, the set of ports $\texttt{ports} \subseteq \texttt{Port}$ that need to be stored in the checkpoint and the number of maximal checkpoints `number`. A program counter $(\texttt{job}, \texttt{mode}, \texttt{time}, \texttt{microTime}) \in \texttt{ProgramCounter}$ describes the exact point within the schedule of a mode $\texttt{mode} \in \texttt{Mode}$ within a job $\texttt{job} \in \texttt{Job}$. For the exact determination of the program counter, the position `microTime` at a logical moment $\texttt{time} \in \mathbb{Q}$ has to be determined. The micro time is a simplified version of the micro time object used in the expanded model and has only three possible values: `BeforeInput`, `BeforeTask`, and `BeforeOutput`.

Error Detection

> *Knowing that a failure has occurred is more important than the actual failure [RM00].*

This quote emphasizes the importance of error detection. Error detection is performed by the specification of tests that check the presence of errors. The literature proposes a variety of tests including control voters, plausibility tests, heartbeats, lifelocks, flow/distributed signature [MM88] and data signatures. Within FTOS, several of these tests are integrated and class inheritance and polymorphism allow an easy addition of further tests.

The abstract syntax of the test base class is $(\texttt{Scheduled}, \texttt{Monitors}) \in \texttt{Test}$ and defines the execution times $\texttt{Scheduled} \subseteq \texttt{ProgramCounter}$, when the tests should be executed, and the set of monitored FCUs $\texttt{Monitors} \subseteq \texttt{FCU}$. For the majority of the tests, this information can be automatically derived during M2M-transformation. Nevertheless, this information can also be explicitly set if tests should only be executed at specific points in time or if the set of monitored FCUs is not obvious.

Port Tests: Many tests focus on the port values as the state of the application is reflected in the concept of ports. In general, one can divide two different types of port tests:

absolute tests and relative tests. Absolute tests determine the status of the monitored FCUs by using a single result, while relative tests compare redundant results. FTOS allows the definition of both tests.

An absolute port test can be defined by (`Scheduled`, `Monitors`, `port`, `min`, `max`, `dev`) $\in$ `TestPortAbsolute`. If no program counter is specified, the test is executed every time the port value is written. Similar, the status of the FCU of the writing actor is evaluated, if no FCUs are specified in `Monitors` explicitly. Using absolute tests, often also called plausibility test, the developer can specify a minimal value `min` $\in \mathbb{Q}$, a maximal value `max` $\in \mathbb{Q}$, and a maximal deviation `dev` $\in \mathbb{Q}$ between two adjacent port values. The success of an absolute port test is determined by the following equation:

$$\begin{aligned}(min = \bot \vee min \leq v'(port)) \wedge (max = \bot \vee max \geq v'(port)) \\ \wedge (dev = \bot \vee dev \geq |v'(port) - v(port)|)\end{aligned} \tag{4.22}$$

where $\mathtt{v}'(\mathtt{port})$ is the new value of the port and $\mathtt{v}(\mathtt{port})$ the old value.

The abstract syntax for relative tests is (`Scheduled`, `Monitors`, `port`, `abs`, `rel`, `stringency`) $\in$ `TestPortRelative`. Instead of one FCU, the test monitors the FCUs of all actors accessing the port simultaneously, if `Monitors` is not specified. The redundant results can be compared using an absolute value `abs` $\in \mathbb{Q}$ or a relative value `rel` $\in \mathbb{Q}$. If both `abs` and `rel` are specified, only one condition must be satisfied. Two results $\mathtt{v_i}, \mathtt{v_j}$ agree on the value of port `p` if:

$$agree(v_i, v_j) = \begin{cases} (max(|v_i|, |v_j|) * rel) \geq |v_i - v_j| & , abs = \bot \\ abs \geq |v_i - v_j| & , rel = \bot \\ (((max(|v_i|, |v_j|) * rel) \geq |v_i - v_j|) \vee (abs \geq |v_i - v_j|)) & , otherwise \end{cases} \tag{4.23}$$

By the attribute `stringency` the developer can define the required rate of result conformance. Possible values are `Majority`, `AbsoluteMajority` and `Unanimity`. For `Majority` all results that agree on the result and form a majority are assumed to be correct. If `stringency` = `AbsoluteMajority`, the number of results that agree on a value must be greater than the number of expected results divided by two. The most stringent type is `Unanimity`, where all expected results must agree on the value.

Let $\mathtt{V_{res}}=\{\mathtt{v_0},\mathtt{v_1},\ldots\mathtt{v_{n-1}}\}$ be the set of expected results for port `port` that should be evaluated. Based on the definition of the `agree` function in Equation 4.23, the set of results $\mathtt{V_i} \subseteq \mathtt{V_{res}}$ that match with $\mathtt{v_i}$ is derived by the following equation:

$$V_i = \{v_j \in V_{res} | agree(v_i, v_j) \tag{4.24}$$

The correctness $\mathtt{c_i}$ of a result $\mathtt{v_i}$ is derived by the following equation:

$$c_i = \begin{cases} \forall v_j \in V_{res} : |V_i| > |V_j| \vee agree(v_i, v_j) & , stringency = Majority \\ |V_i| = n \vee (\exists v_j \in V_{res} : |V_j| > |V_{res}|/2 \wedge agree(v_i, v_j)) & , stringency = Unanimity \\ |V_i| > n/2 \vee (\exists v_j \in V_{res} : |V_j| > |V_{res}|/2 \wedge agree(v_i, v_j)) & , otherwise \end{cases} \tag{4.25}$$

This definition accepts also border cases in the sense that two values that are assumed to be correct agree may not agree with each other. If this behavior is undesirable, one can easily define new values for the `stringency` attribute and define appropriate test implementations.

Liveness Tests: The second important class besides tests on ports is the class of liveness tests. Liveness tests are based on periodical heartbeat signals, messages send from one component to another to signal liveness. In FTOS, heartbeats are implemented at the level of ECU components. Using a liveness test object $(\texttt{Scheduled}, \texttt{Monitors}, \texttt{ecu}, \texttt{evalPeriod}) \in \texttt{TestLiveness}$, the developer can specify the ECU `ecu` that should be tested. The test is performed on every ECU affected by the outcome of the liveness test (see Section 4.6.3). To save bandwidth, system messages are considered for this test. The observed unit checks at every program point `time` whether a message was send to the observing nodes during the elapsed evaluation interval `evalPeriod`. Only if no message was sent to one of the observing units, an additional heartbeat signal is send. The observing units simply check the existence of a message during the evaluation interval, when the system reaches a program counter defined in `Scheduled`.

Tests Based on Application Logic: Most of the fault-tolerance frameworks support the detection of faults by monitoring the behavior of replicas. However, tests that require application logic cannot be integrated or require an adaptation of the application code [KBWI99]. An example of useful application dependent tests are tests based on control flow signatures [MM88]. Control flow signatures define a sequence of correct messages/results. To test these sequences, a n-step history of the system state is required. To reduce the amount of required memory, a sophisticated analysis is necessary. Another problem is that the definition of correct sequences based on the whole system state is usually difficult. FTOS tackles both issues by allowing the definition of user defined tests. These tests can easily be modeled using state machines and the code generator can produce efficient code. The abstract syntax is $(\texttt{time}, \texttt{Monitors}, \texttt{Read}, \texttt{Write}, \texttt{statemachine}) \in \texttt{TestUserDefined}$. The ports that are accessed by the state machine are specified by a set of ports `Read`. Internal results can be written to a set of ports `Write`, which are only allowed to be used by the state machine. The execution of the state machine is invoked at every logical time `time`. As user defined tests can monitor arbitrary FCUs, at least one FCU must be specified in `Monitors`. The status of the referred FCUs can be accessed and written analog to the ports in `Read` and `Write`.

Error Treatment

The online error treatment mechanisms reflect the concept of correctors as defined by Arora and Kulkarni [AK98b]. In contrast to the application-dependent approach of Arora and Kulkarni, FTOS provides generic error treatment operations. The correctors proposed by Arora and Kulkarni are again transitions that are triggered by the activation of a guard, in the context of this thesis a test result. The transitions update the system state by updating the values of some variables. The variables reflect both application variables and the program counter.

Therefore, two basic operations can be defined to achieve fault-tolerance: port / trigger result updates and changes of the program counter. The new position, where execution of the system should be continued can be specified using a program counter $\texttt{pc} \in \texttt{ProgramCounter}$. A port update $(\texttt{p}, \texttt{function}, \texttt{Reads}) \in \texttt{PortUpdate}$ specifies a

function `function` that is used to calculate a new value for the port `p`. The function can be specified analog to transition functions in state machines and is based on the values of the ports `Reads`. To guarantee determinism the ports contained in `Reads` are not allowed to be updated at the same time.

Based on these two basic fault-tolerance mechanisms, it is possible to define generic fault-tolerance mechanisms that reflect the standard mechanisms known from literature. The currently supported mechanisms are discussed in the following.

Replica Management: Passive and semi-active replication is supported by the definition of a replica management object. The abstract syntax for such an object is $(\texttt{parallel}, \texttt{min}, \texttt{failsafe}, \texttt{Updates}) \in \texttt{ReplicaManagement}$. A replica management object automates the consistent determination of a master for a parallel execution schedule $\texttt{parallel} \in \texttt{ParallelExecution}$. The decision is computed every time tests are performed that check the correctness of the associated FCUs. The realization of the replica management is based on triggers. In Section 4.4.4, the transformation of the parallel execution schedule into a sequential schedule and the generation of an according trigger was discussed. The trigger is based on a newly generated port that contains the ID of the master.

Every time a status of the involved FCUs changes, the master is computed again. The selection of the master depends on the implementation, but it is guaranteed to be unique for the whole system. In addition to the selection of a new master, the system excludes the erroneous replica for error recovery, if the FCU is related to an ECU.

For semi-active replication, the developer has to decide whether safety or reliability should be achieved. If safety is the main goal, the system switches into fail-safe mode as soon as some errors are detected. Instead, a reliable system tries to keep the system running until the last replica fails. Whether safety, reliability, or a combination of both is targeted can be specified by the `min` attribute. `min` specifies the minimal number of components that must be correct to continue the system's execution in the current application mode. If the number of correct replicas falls below `min`, the system is either stopped or continued in fail safe-mode. The fail-safe mode is defined by $\texttt{failsafe} \in \texttt{ProgramCounter}$ and specifies the program counter where the system execution should continue its operation. In addition, it might be necessary to update some ports. The relevant information can be specified by the use of port updates $\texttt{Updates} \subseteq \texttt{PortUpdate}$.

Rollback: Rollback operations are obviously simple port updates and a change of the program counter. However, the values for the ports and the new program counter cannot be determined at development phase, but depend on the previous state, which is stored within a checkpoint. The definition of a rollback operation $(\texttt{Trigger}, \texttt{Corrects}, \texttt{checkpoint}, \texttt{Test}_\texttt{r}, \texttt{failsafe}, \texttt{Updates}) \in \texttt{Rollback}$ defines the relevant checkpoint object `checkpoint` and the fault configuration sets $\texttt{Trigger} \subseteq \texttt{FaultConfigurationSet}$. Every time, a fault configuration set $\texttt{fc} \in \texttt{Activation}$ becomes active in the system, the system is rolled back to the last checkpoint created by $\texttt{checkpoint} \in \texttt{Checkpoint}$. The operation is assumed to repair some system state. The success of the rollback operation can be checked by additional tests specified by $\texttt{Test}_\texttt{r} \subseteq \texttt{Test}$. The FCUs that are assumed to be repaired by the

rollback operation are stated in $\texttt{Corrects} \subseteq \texttt{FCU}$. All test results associated with these fault containment units are reset, if the additional tests are successfully evaluated. If in contrast the additional tests fail, the rollback operation is repeated.

If no checkpoint is available that can be used for the rollback operation, the system is either stopped or switch to a fail-safe mode if available. The realization is analog to the mechanisms used for replica management.

User Defined Reaction: Finally, some applications might need application-specific mechanisms beyond the generic mechanisms previously presented. Examples are graceful degradation or rollforward. To specify these mechanisms, FTOS provides the user defined treatment class. The abstract syntax is $(\texttt{Trigger}, \texttt{Previous}, \texttt{Time}, \texttt{continue}, \texttt{Updates}, \texttt{Excludes}) \in \texttt{TreatmentUserDefined}$. The reaction is triggered whenever the active fault configuration set changes from one set specified in $\texttt{Previous} \subseteq \texttt{FaultConfigurationSet}$ to a set in $\texttt{Trigger} \subseteq \texttt{FaultConfigurationSet}$. The definition of the previously active fault configuration set allows to define both upgrade and downgrade operations. Using the set of program counters $\texttt{Time} \subseteq \texttt{ProgramCounter}$, the reaction can be made dependent on the current program counter. The new program counter, where the system should continue the operation is specified by $\texttt{continue} \in \texttt{ProgramCounter}$. If $\texttt{continue} = \perp$, the job is stopped. Analog to the other mechanisms, the `Updates` reference allows the definition of functions to update port values. In addition, the developer can specify FCUs that should be excluded from operation to perform fault recovery.

Error Recovery

The term error recovery comprises all mechanisms in FTOS that are performed offline to recover from a recently detected error. The recovery is performed in two phases: recovery and integration.

Recovery: The mechanisms for error recovery are described for each FCU by defining a recovery object. It is important to note that only FCUs referring to ECUs can be excluded. All other units must be excluded by changing to another schedule. The abstract syntax of this class is $(\texttt{fcu}, \texttt{Action}, \texttt{Test}_\texttt{r}, \texttt{Integration}) \in \texttt{ErrorRecovery}$. Each object describes the actions $\texttt{Action} \subseteq \texttt{RecoveryAction}$ performed to recover the FCU $\texttt{fcu} \in \texttt{FCU}$. The recovery actions are very often application dependent and range from a simple restart of the program to the transmission of an email to the maintenance personnel. FTOS provides only the basic mechanisms, but offers an interface to define application-specific actions. A recovery action $(\texttt{action}, \texttt{max}, \texttt{function})$ is described as the combination of an action $\texttt{action} \in \texttt{ActionType}$, the maximal error count `max` and an optional function name `function`. Possible values of `action` are `Reboot`, `Restart`, `Shutdown`, `Ignore` and `UserDefined`. `Reboot` leads to the reboot of the FCU. `Restart` denotes the software restart of the according component and `Shutdown` simply leads to the stop of the execution. `Ignore` is in a sense not a real recovery operation and assumes that the system will recover when continuing the execution. Chapter 5 discusses the use of the `Ignore` operation.

If several recovery actions are specified, the selection depends on the attribute `max`. For each FCU with more than one recovery action, an error counter is implemented. The

run-time system selects the action with the lowest `max` value that is greater or equal than the current error count. The counter is only reset if the ECU is rebooted. A more sophisticated error counter, which considers for instance also the intervals between different errors, can be added easily.

The successful completion of the recovery action is checked by the tests specified in $\texttt{Test}_\texttt{r} \in \texttt{Test}$. If successful, the run-time system proceeds with the integration into the running system.

Integration The integration of repaired or new components is complicated because of the *amnesia* state [Pow94]. The components have no knowledge of the system state and therefore a state synchronization has to be performed. The literature [PCD91, Rus96, BGG+98] points out that the design of a generic state synchronization is difficult, since the state is very application specific. Therefore, most fault-tolerance approaches such as DEAR-COTS do not support integration at system level [PVW04].

However, all objects that contain some state information are specified explicitly in the models of FTOS. The developer can determine the exact information required for a successful integration by specifying integration objects. The abstract syntax is $(\texttt{P}_\texttt{i}, \texttt{Trigger}_\texttt{i}, \texttt{Test}_\texttt{i}, \texttt{Pre}, \texttt{Entry}, \texttt{continue}, \texttt{Updates}, \texttt{Reset}) \in \texttt{Integration}$. Relevant information can be contained in ports, in triggers, and in test results. The values of the ports $\texttt{P}_\texttt{i} \subseteq \texttt{Port}$ are required to get a consistent application state, the results of the triggers $\texttt{Trigger}_\texttt{i} \subseteq \texttt{Trigger}$ to guarantee a correct execution, and the results of tests $\texttt{Test}_\texttt{i}$ to guarantee the test consistency and a consistent determination of the current fault configuration.

Furthermore, the developer can restrict the integration by specifying a set of fault configuration sets $\texttt{Pre} \subseteq \texttt{FaultConfigurationSet}$ and a set of program counters $\texttt{Entry} \subseteq \texttt{ProgramCounter}$. Only if one fault configuration set $\texttt{fc} \in \texttt{Pre}$ is active and the system is at one program counter $\texttt{pc} \in \texttt{Entry}$, the integration can be performed. A requirement for a successful integration is that at the time of integration no relevant task is executing within the running system that may influence the success of the integration negatively. This condition is checked during model validation.

Regarding the effect of integration, FTOS distinguishes between silent and non-silent integration. In silent integration, the running system simply forwards the required information including the current program counter. The integrating unit can vote on the incoming information, if some information is sent from more than one unit. If successful, the integrating ECU can start participating in the execution, but is first not recognized as repaired. The real integration is done during the subsequent tests, when the running system updates the test results reflecting the state of the integrating unit. The other possibility is non-silent integration. A non-silent integration influences the execution of the system directly and requires that the reaction of the system is consistent. Therefore, the non-silent integration consists of two phases [PABD+99]. The first phase is analog to the silent-integration. However, the running system waits for the integrating unit to signal the integration success. If successful, the system may change the program counter to $\texttt{continue} \in \texttt{ProgramCounter}$, update some port values according to $\texttt{Updates} \subseteq \texttt{PortUpdate}$ and reset the test results $\texttt{Reset} \subseteq \texttt{Test}$ that are related

to the integrating unit. The non-silent integration can be used to reverse a preceding graceful degradation.

4.6.3 Model-To-Model Transformation in the Context of the Fault-Tolerance Model

This section discusses some important aspects of the M2M transformation in the context of the fault-tolerance model. The transformation computes for example the points in time for the execution and the monitored FCUs for each test, if this information is not specified by the developer. The relevant transformations are straightforward and omitted here. Further transformation rules compute the set of relevant fault-tolerance mechanisms for each ECU e. This set contains all relevant pro-active operations, tests, treatment operations, and recovery actions. This section illustrates the transformation using the computation of the relevant treatment operations for one unit.

An ECU e is affected by an error treatment, if the behavior (schedule and ports read by actors executed on e) of the ECU is influenced by the result of a test. Replica management objects affect the execution of a control unit if an actor is directly affected, if a fail-safe mode is specified and the job is also executed on the ECU, or if a relevant port is updated. Similarly, a rollback operation affects the associated job and therefore the ECUs executing this job. The only exceptions are rollback operations with a checkpoint that is executed at only one point in the schedule. In this case, only some ports may be updated and hence only ports are affected that read some of these ports. A user defined reaction is added similar to the other treatment objects if it affects a relevant port or job. The set of error treatment objects is derived by the following equation:

$$
\begin{aligned}
&ErrorTreatment_e = \\
&\quad \{rm \in ReplicaManagement \mid rm.parallel.type = ColdRedundancy \wedge \\
&\qquad \exists a \in ActorReference, \exists s \in rm.parallel : a.actor.ecu = e \wedge s.contains(a)\} \cup \\
&\quad \{rm \in ReplicaManagement \mid rm.parallel.type = HotRedundancySingleOutput \wedge \\
&\qquad \exists o \in OutputReference, \exists s \in rm.parallel : o.actor.ecu = e \wedge s.contains(o)\} \cup \\
&\quad \{rm \in ReplicaManagement \mid rm.failsafe \neq \bot \wedge \\
&\qquad contains(e.Jobs, rm.failsafe.job)\} \cup \\
&\quad \{rm \in ReplicaManagement \mid rm.Updates \neq \bot \wedge \\
&\qquad \exists up \in rm.Updates : \ contains(up.p.ReadEcus, e)\} \cup \\
&\quad \{rb \in Rollback \mid \ |rb.checkpoint.Scheduled| > 1 \wedge \\
&\qquad contains(e.Jobs, rm.scheduled.get(0).job)\} \cup \\
&\quad \{rb \in Rollback \mid \ |rb.checkpoint.Scheduled| = 0 \wedge \\
&\qquad \exists p \in rb.Ports : \ contains(p.ReadEcus, e)\} \cup \\
&\quad \{rb \in Rollback \mid rb.failsafe \neq \bot \wedge \\
&\qquad contains(e.Jobs, rb.failsafe.job)\} \cup \\
&\quad \{rm \in Rollback \mid rb.Updates \neq \bot \wedge \\
&\qquad \exists up \in rb.Updates : \ contains(up.p.ReadEcus, e)\} \cup \\
&\quad \{ud \in TreatmentUserDefined \mid \ contains(e.Jobs, ud.Time.get(0).job\} \cup \\
&\quad \{ud \in TreatmentUserDefined \mid \ \exists p \in ud.Ports : \ contains(p.ReadEcus, e)\}
\end{aligned}
\tag{4.26}
$$

Based on the set $\texttt{ErrorTreatment}_\texttt{e}$, the relevant set of fault configuration sets $\texttt{FC}_\texttt{e}$ can be computed. The set contains all fault configuration sets that are referenced by the error treatment object. The set $\texttt{FC}_\texttt{e}$, in turn, allows the calculation of the relevant FCUs $\texttt{FCU}_\texttt{e}$. Finally, the set of relevant tests $\texttt{Test}_\texttt{e}$ is calculated by the tests monitoring the FCUs in $\texttt{FCU}_\texttt{e}$. In addition, the set of ports is expanded with additional ports that are used to save the information of the different checkpoints.

4.6.4 Operational Semantics

Based on the information contained in the fault and fault-tolerance model, the schedule calculated during the M2M transformation of the software model is expanded by operations related to fault-tolerance mechanisms. This section defines the additional phases and discusses the concrete actions.

The implementation is performed in four phases reflecting the error detection (phase 1), error treatment (phase 2), integration (phase 3), and pro-active operations (phase 4). These four phases are inserted again at three points in the schedule according to the value of the `microTime` attribute of the related program counter: before input related schedule operations, before task related schedule operations or before output related schedule operations. The resulting schedule phases are depicted in Figure 4.21.

In the following, the different phases are formally specified. Without loss of generality, the description is based on the fault-tolerance mechanism phase before the output phase. The other phases can be directly derived from this description.

As a reminder, the state of a job `j` on an ECU `e` is described by the program configuration $\texttt{C}_{\texttt{j},\texttt{e}} = (\texttt{m}, \delta, \mu\delta, \texttt{v}, \texttt{trigger}, \tau)$. The relevant local schedule for mode `m` on ECU `e` was defined by $\texttt{sched}_{\texttt{e},\texttt{m}}$. In the context of the fault-tolerance mechanisms, this configuration is expanded by the valuation function $\texttt{test} : \texttt{Test} \times \texttt{FCU} \rightarrow \mathbb{B}$ that expresses the result of a test with respect to the monitored FCUs. `True` represents the correctness assumption, `False` the assumption that the FCU is erroneous. In addition, the currently assumed fault configuration set is reflected in $\texttt{fc} \in \texttt{FaultConfigurationSet}$. Therefore, the expanded program configuration is $\texttt{C}_{\texttt{j},\texttt{e}} = (\texttt{m}, \delta, \mu\delta, \texttt{v}, \texttt{trigger}, \texttt{test}, \texttt{fc}, \tau)$.
It is important to note that `test` and `fc` only reflects the system assumptions on the fault state of the system and may not match the real fault state. This implies that the system steps described in Section 4.4.5 have no influence on `test` and `fc`.

The micro time $\mu\delta$ assigned for the individual steps is of course changed by the introduction of the fault-tolerance mechanism phases. In the following, the order of the sequence is based on the order in Figure 4.21.
The operation of the different fault-tolerance steps can be described as follows:

1. [**1st Test Phase**, $\mu\delta = 4$]: Let $\texttt{T} = \{\texttt{t} \in \texttt{Test}_\texttt{e} | \exists \texttt{pc} \in \texttt{t.Scheduled} : \texttt{pc.mode} = \texttt{m} \wedge \texttt{pc.time} = \delta \wedge \texttt{pc.microTime} = \texttt{BeforeOutput}\}$ be the set of tests that need to be executed at that logical moment. Let

$$test'(t, f) = \begin{cases} t.function()(f) & , \exists t \in T \wedge f \in t.Monitors \\ test(t, f) & , otherwise \end{cases} \tag{4.27}$$

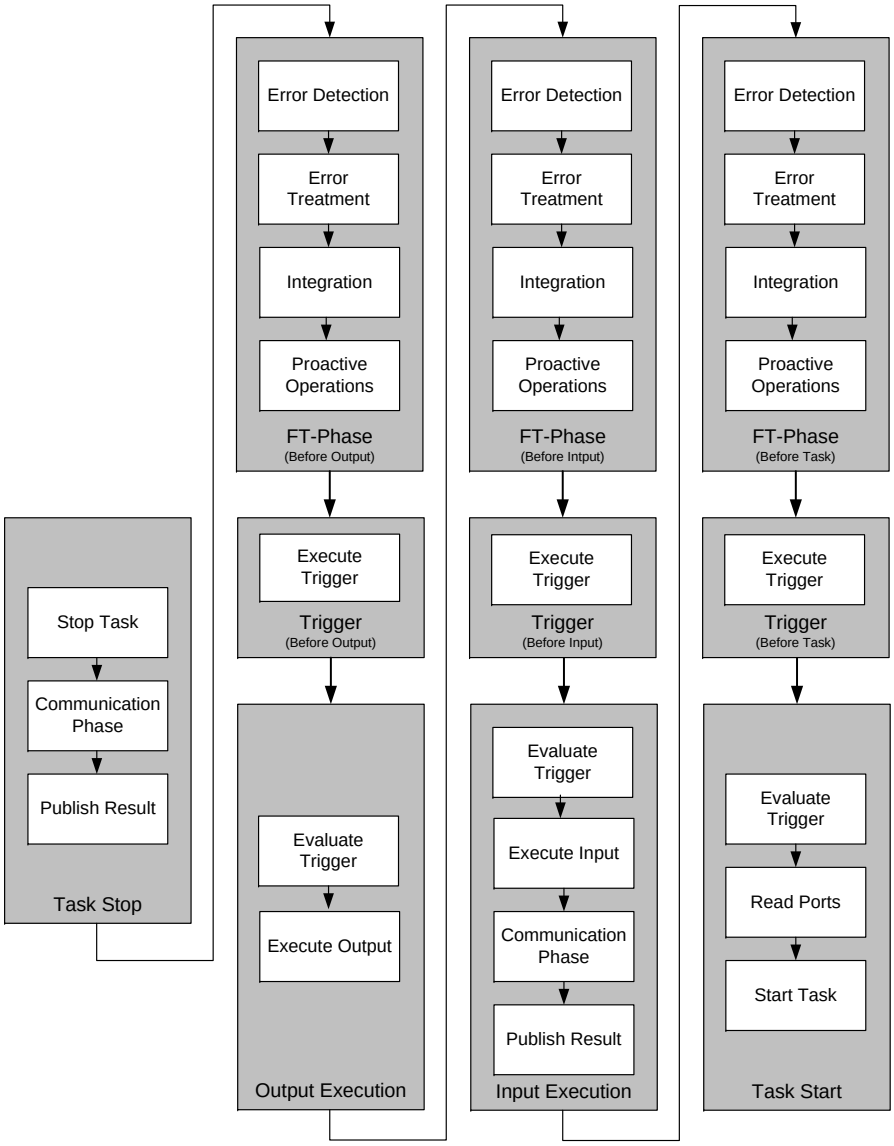


Figure 4.21: Additional Schedule Phases for Fault-Tolerance Mechanisms

be the test valuation function after the performance of the tests. The parameters for a specific test function depend on the type of the test and are omitted here for simplicity reasons. The successor configuration is obtained by $\mathtt{C}'_{\mathtt{j,e}} = (\mathtt{m}, \delta, 5, \mathtt{v}, \mathtt{trigger}, \mathtt{test}', \mathtt{fc}, \tau)$.

2. [**1st Treatment Phase**, $\mu\delta = 5$: Let `fcu` be a function to determine the status of all FCUs $\mathtt{f} \in \mathtt{FCU_e}$ with $\mathtt{fcu(f)} = \bigwedge_{\forall \mathtt{t}:\mathtt{f} \in \mathtt{t.Monitors}} \mathtt{test(t,f)}$. Let

$$\begin{aligned} FC = \{ & fc \in FaultConfigurationSet | \\ & \forall f \in fc.Correct: \ fcu(f) \wedge \\ & \forall f \in fc.Erroneous: \ \neg fcu(f) \wedge \\ & \forall im \in fc.Implicit: \ im.min <= |\{f \in im.FCUs | fcu(f)\}| <= im.max\} \end{aligned} \tag{4.28}$$

be the active fault configurations set. If $\mathtt{FC} = \emptyset$, the actual number of faults exceeds the assumptions specified in the fault hypothesis and the system execution is aborted by setting $\mu\delta' = -1$. Otherwise, the set must contain exactly one element, due to the requirement that the fault configuration sets are not overlapping. Let $\mathtt{fc}'$ be this element. If $\mathtt{fc}' \neq \mathtt{fc}$, the fault configuration set has changed and an error treatment must be initiated. The validation rules demand that for a certain combination of previous configuration `fc`, current configuration $\mathtt{fc}'$ and current program counter `pc` at most one treatment object $\mathtt{tr} \in \mathtt{ErrorTreatment}$ exists. Without loss of generality, `tr` is assumed to be of type `TreatmentUserDefined`. Let δ' be `tr.pc.time` if a new fault configuration set is active and `tr` is defined and $\delta' = \delta$ otherwise. Let $\mathtt{m}'$ be the new mode `tr.pc.mode` if a new fault configuration set is active and there exists a treatment operation `tr` and otherwise $\mathtt{m}' = \mathtt{m}$.
$\mu\delta'$ can be derived from the following equation:

$$\mu\delta' = \begin{cases} tr.pc.time, & fc' \neq fc \wedge tr \neq \bot \wedge \neg(e.fcu \in tr.Excludes) \\ -1, & fc' \neq fc \wedge \neg(e \in tr.Excludes) \vee FC = \emptyset \\ 6, & otherwise \end{cases} \tag{4.29}$$

It is important to mention that `tr.pc.microTime` is adapted from the simplified micro time model in the fault-tolerance model to the augmented micro time model during M2M transformation. The abort of the system operation is followed by recovery operations as specified in the fault-tolerance model. As these operations are not executed within the system operation and are typically user-defined, they are not formalized. Let

$$v'(p) = \begin{cases} u.function(v(u.Reads))(p) & , tr \neq \bot \wedge \exists u \in tr.Updates : p = u.port \\ v(p) & , otherwise \end{cases} \tag{4.30}$$

be the valuation function that agrees in case of a new fault configuration set with all port updates described in `tr` and with `v(p)` otherwise. The new configuration is determined by $\mathtt{C}'_{\mathtt{j,e}} = (\mathtt{m}', \delta', \mu\delta', \mathtt{v}', \mathtt{trigger}, \mathtt{test}, \mathtt{fc}', \tau)$.

3. [**1st Integration Phase**, $\mu\delta = 6$]: Let $\texttt{IP} = \{\texttt{ip} \in \texttt{Integration} \mid \texttt{contains}(\texttt{Pre}, \texttt{fc}) \wedge (\exists \texttt{t} \in \texttt{entry} : \texttt{t.mode} = \texttt{m} \wedge \texttt{t.time} = \delta \wedge \texttt{t.microTime} = \mu\delta)\}$ the set of possible integration objects. If $\texttt{IP} \neq \emptyset$, the system checks whether an integration request $\texttt{ip} \in \texttt{Integration}$ was registered in the network component. If an open request is pending, the system sends the values of the required ports $\texttt{ip.P}_\texttt{i} \cap \texttt{P}_\texttt{e}$ available on the ECU, the decisions of triggers $\texttt{ip.Trigger}_\texttt{i} \cap \texttt{Trigger}_\texttt{e}$, the results of tests $\texttt{ip.Test}_\texttt{i} \cap \texttt{Test}_\texttt{e}$, and the program counter $\texttt{p}_\texttt{c}$ to the integrating unit $\texttt{i}$. The integrating unit compares the different system configurations and adopts the result. A successful integration is signaled to the network component by the signal $\texttt{ip.success}$ for non-silent integration. The system supports the integration of only one ECU at a time, since this is sufficient for nearly all fault-tolerant systems. Let $\texttt{FCU}_\texttt{i}$ be the set of all FCUs associated with $\texttt{i}$ (FCUs of all hardware components contained in $\texttt{i}$ and actors executed on $\texttt{i}$). Let

$$test'(t,f) = \begin{cases} true & ,t \in Reset \wedge contains(FCU_i, f) \wedge f \in t.Monitors \wedge \\ & \quad ip.success \\ test(t,f) & ,otherwise \end{cases} \tag{4.31}$$

be the status determination function that resets all test results for tests contained in $\texttt{Reset}$ that reference an FCU associated with $\texttt{i}$. Let

$$\begin{aligned} FC = \{fc \in FaultConfigurationSet \mid & \\ & \forall f \in fc.Correct : \ fcu(f) \wedge \\ & \forall f \in fc.Erroneous : \ \neg fcu(f) \wedge \\ & \forall im \in fc.Implicit : \ im.min <= |\{f \in im.FCUs | fcu(f)\}| <= im.max\} \end{aligned} \tag{4.32}$$

be the new active fault configurations set. Due to the requirement that only one fault configurations set can be active at a time, the set consists of exactly one element. Let $\texttt{fc}' = \texttt{FC.get}(0)$ be this element, if the non-silent integration was successful and $\texttt{fc}' = \texttt{fc}$ otherwise. Let

$$v'(p) = \begin{cases} u.function(v(u.Reads))(p) & ,\exists u \in i.Updates : p = u.port \wedge ip.success \\ v(p) & ,otherwise \end{cases} \tag{4.33}$$

be the valuation function that updates the port values in case of a successful non-silent integration. For a successful non-silent integration, let $\texttt{m}'$ be the new mode as specified in $\texttt{ip.consec.mode}$ if $\texttt{ip.consec} \neq \perp$, δ' be the new time that is $\texttt{ip.consec.time}$ if $\texttt{ip.consec} \neq \perp$ and let $\mu\delta$ be $\texttt{ip.consec.microTime}$ respectively $\texttt{7}$. The successor configuration is obtained by $\texttt{C}'_{\texttt{j,e}} = (\texttt{m}', \delta', \mu\delta', \texttt{v}, \texttt{trigger}, \texttt{test}', \texttt{fc}', \tau)$ for non-silent and by $\texttt{C}'_\texttt{j}, \texttt{e} = (\texttt{m}, \delta, \texttt{7}, \texttt{v}, \texttt{trigger}, \texttt{test}, \texttt{fc}, \tau)$ for silent integration.

4. [**1st Proactive Operation Phase**, $\mu\delta = 7$]: Let $\texttt{CP} = \{\texttt{cp} \in \texttt{CheckPoint} \mid \texttt{cp.mode} = \texttt{m} \wedge \texttt{cp.time} = \delta \wedge \texttt{t.microTime} = \mu\delta)\}$ be the set of current checkpoints. For each checkpoint $\texttt{cp} \in \texttt{CP}$, let $\texttt{P}_\texttt{cp}$ be the ports that were allocated during M2M

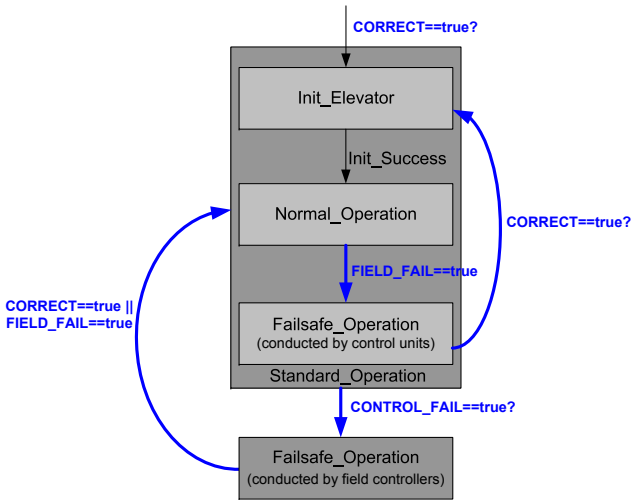


Figure 4.22: Elevator - Interaction of Software Model and Fault-Tolerance Mechanism

transformation to store the different checkpoints. For simplification reasons, let $\mathtt{f_{cp}} : \mathtt{Ports_e} \times \mathtt{Mode} \times \mathbb{Q} \rightarrow \mathtt{P_{cp}}$ be the storage function that saves the relevant ports, the currently executed mode and the logical time to the checkpoint ports. Let $\mathtt{v}'$ be the valuation function that agrees for each checkpoint $\mathtt{cp}$ with $\mathtt{f_{cp}}$ on the values of $\mathtt{P_{cp}}$ and with $\mathtt{v}$ otherwise. The successor configuration of this step is specified by $\mathtt{C}'_{\mathtt{j,e}} = (\mathtt{m}, \delta, \mathtt{8}, \mathtt{v}', \mathtt{trigger}, \mathtt{test}, \mathtt{fc}, \tau)$

4.6.5 Fault-Tolerance Mechanism in Example Applications

This section gives an overview on the used mechanisms for the demonstrators. The application controlling the rod uses a relative test $(\mathtt{Result}, \epsilon, -\mathtt{1}, \mathtt{AbsoluteMajority})$ to detect errors in the value domain. Due to inconsistencies caused by the analog measurement and small synchronization errors, a bit-by-bit comparison is not applicable. This problem can be solved by defining a small toleration interval ϵ. As test stringency, absolute majority is selected. Furthermore, a liveness test has to be defined to detect fail-stop failures. The error treatment is based on replica management. The parallel execution of the software model is referenced. At least two elements have to be available. The error recovery is based on rebooting the unit. The integration into the running system is performed at the logical time $\mathtt{0}$ before the input operation.

Figure 4.22 demonstrates the concept in the context of the elevator example. In section 4.4, the different modes are described. The motivation to group the three sub modes `Init`, `Normal` and `FieldFailure` into one mode is to avoid unnecessary tests within the field controllers as their behavior is not changed in one of these modes. However to realize this grouping, it is necessary to signal the existence of failures of the field controllers to the application logic. This was initially not foreseen due to the strict separation of application logic and fault-tolerance mechanisms. By defining an user defined fault-tolerance mechanism, this is however possible. Whenever the active fault configuration is switched to `FieldFailure`, the system performs an user defined

fault-tolerance mechanism that simply updates a special port `FieldFailure` to `1` that is interpreted by the task function. Vice versa, whenever all field controllers are assumed to be correct, the system updates the port to `0`.

Another user defined fault-tolerance mechanism realizes the mode switch if both control units fail. Alternatively one could also use a replica management object with 1 as minimal number and a fail-safe mode. Finally, only the fault configuration `Correct` is marked as startable. This adds a condition to the system start, as depicted in Figure 4.22.

The used tests are all liveness tests. Due to the optimization of the relevant FCUs to determine the active fault configuration, the field controllers only have to monitor the two control units. In contrast, the two control units must monitor all ECUs because they are affected by the port updates.

The importance of the different integration types can be pointed out in the elevator application. In general, the integration is performed silently. However, in one case a non-silent integration is required. If both control units previously failed, the system should start immediately with its standard operation if a control unit can successfully integrate. The relevant program counter is set to the begin of the `Standard_Operation` schedule. The associated liveness test of the integrating unit is reset. A port update sets the variables that are only available on the control units to the initial values.

4.7 Summary

This summarizes the main concepts of the meta-model provided by FTOS. The meta-model consists of four sub meta-models used to describe the hardware architecture, the software components and their interaction, the expected faults, and the fault-tolerance mechanisms.

The hardware model identifies the hardware components and the network topology. It is used to select the run-time system. By exploiting the object-oriented concepts class inheritance and polymorphism, the expandability of the hardware model with respect to additional hardware devices and communication protocols is guaranteed.

The software model guarantees determinism by the use of logical execution times (LET) and global ports. The concept LET hides the physical execution and its low-level implementation. Global ports hide the realization of the distributed execution of the software components and provide means to specify consensus requirements in an abstract way. The concepts of jobs and modes allow the specification of sophisticated execution schemes.

The fault model forces the developer to specify the fault assumptions in a formal way. In contrast to other approaches, the fault hypothesis is not restricted to a limited number of faults and fault types. Instead of relying on the specification of faults, FTOS uses the more general concept to describe faults by their application-independent effect. Therefore, the fault model can be used both for the correct implementation of the fault-tolerance mechanisms and the verification of the system.

The concrete fault-tolerance mechanisms that should be applied in the system are de-

fined in the fault-tolerance model. FTOS uses four building blocks to define the fault-tolerance mechanisms, namely error detection, online error treatment, offline error recovery, and pro-active operations. This concept allows the composition of different components to realize classical fault-tolerance mechanisms, such as hot-standby, triple-modular redundancy, or rollback recovery. On the other hand, also sophisticated application specific fault-tolerant mechanisms can be modeled with a very high abstraction level without compromising the possibility of extensive code generation.

The approach can be used for both developers with expertise in fault-tolerant systems and novices. Experts benefit especially from the extensive code generation, which is pointed out in Chapter 5. The major advantage for novices is the high abstraction level and the formulation of the different configuration possibilities in the models. In this sense, the model points out the typical pitfalls and provides the solutions. The ease of learning the different concepts was proved by using the tool FTOS in the lab course Real-Time Systems. The students had to implement a simplified version of the elevator example. Although the concepts of FTOS were only discussed briefly, all students were able to implement a running application within 90 minutes.

CHAPTER 5

Realization

Contents

This chapter discusses the realization of the different concepts introduced in the meta-model. The literature proposes various implementations for the different components required in fault-tolerant real-time systems [Pra96]. Therefore, it is not the goal of this thesis to invent new mechanisms, but to point out how these mechanisms can be implemented in a generic way exploiting the concepts of the used modeling language. The chapter starts with a brief overview on the run-time system architecture, the individual components and their roles. In the following, three components are discussed in more detail: the communication component, the fault-tolerance component, and the scheduling component. The chapter is concluded with a discussion of the available prototypes and the experiences made with the demonstrator applications.

5.1 Run-Time System Architecture

Figure 5.1 illustrates the generic architecture of a run-time system for an ECU and identifies the principal components. The arrows represent the data flow between the different components.

The central component is the **system logic** that coordinates the system execution. From a functional point of view, the main task is to logically start and stop the application tasks according to the specified schedule, to execute I/O operations, and to trigger the

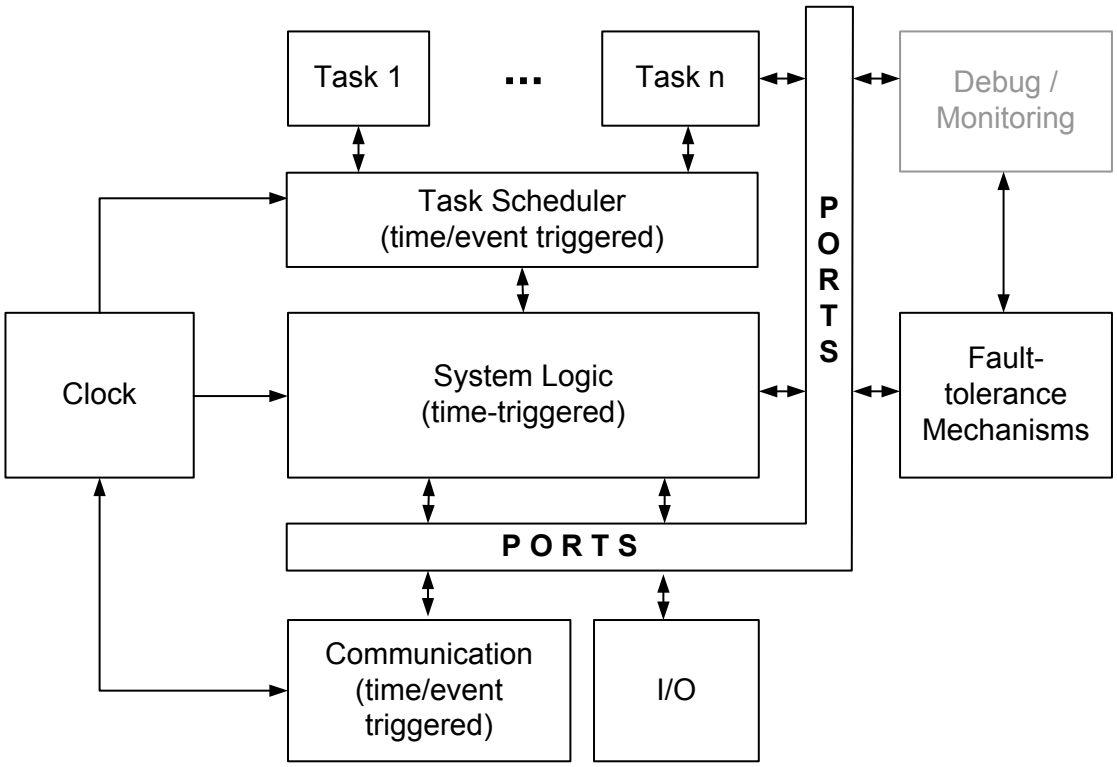


Figure 5.1: Run-time System Architecture

network communication. Mode switches and triggers must be evaluated to select the correct set of actors that have to be executed. The execution of the system logic is performed time-triggered and reflects the timing constraints expressed in the model. The realization of the system logic is straightforward. The sequential schedule computed during M2M transformation can be mapped directly to the related operations. All possible branches are known in advance. Hence, the implementation is analog to the code synthesis for a finite state automaton.

To achieve fault-tolerance, the system logic has to monitor the status of the relevant fault containment units and needs to trigger appropriate actions, if an error is detected or a new/repaired unit is integrated. The functionality for the fault-tolerance mechanisms is provided by the **fault-tolerance component**. This component is explained in detail in Section 5.3.

A **clock** component provides the local time base. The clock synchronization is realized by the communication component.

The application **tasks** are realized as separate components encapsulating the application functionality. The system logic performs the logical task start and task stop by copying the parameters from the global ports to the task's local ports and vice versa. A **port** component realizes the management and allocation of these variables. Besides the ports specified in the software model, several additional ports are added to realize the communication between the run-time components.

The actual physical assignment of the CPU to a task is realized by the **scheduling** component. The different possibilities to realize this component are discussed in detail in Section 5.4. However, it is important to note that although the logical execution is performed time-triggered, the scheduling component can be implemented either time- or event-triggered [HHK03].

The interaction with the environment is performed by **I/O** components that realize all I/O operations that are not related to network operations. The I/O components are passive in the way that they are triggered by the system logic. The communication is realized by an own component, the **communication** component. Besides the network-related I/O operations, this component also realizes the communication within the distributed system. Analog to the scheduling component, this component can be implemented using either time- or event-triggered communication protocols. The realization is discussed in more detail in Section 5.2.

Finally, a **debug/monitoring** component, usually not integrated in the run-time system, can be added for monitoring purposes. This component is designed in a way that the system's behavior is not influenced. It can be used to log the system execution.

5.2 Communication Component

The communication component has three tasks: besides its obvious role to realize the communication within the distributed system and with the environment, it has to perform the clock synchronization and to manage liveness signals. The latter task is discussed in Section 5.3.

The communication within the distributed system can be performed both event- and time-triggered. Based on the information contained in the model, it is possible to compute a time-triggered communication schedule [Far06]. Nevertheless, FTOS supports also event-triggered protocols to minimize the restrictions. It is important to point out that even for an event-triggered design of the communication component, knowledge about the send and receive events is available due to the time-triggered execution of the system logic that triggers the send operation. The only requirement on the communication protocol is that the message latency has an upper bound for a given network load. This condition is satisfied by most standard communication protocols used in the domain of embedded system such as Switched Ethernet or Controller Area Network (CAN). Templates to support both protocols are available; an integration of the time-triggered protocol (TTP) [TTT03] into FTOS is planned.

The second task of the communication component is clock synchronization. Clock synchronization is one of the key features required to design a fault-tolerant system [WLG+78]. Two conditions must be fulfilled by a clock synchronization algorithm [PABD+99]:

- The **agreement** condition requires that the skew between the local time of non-faulty nodes is bounded.
- The **accuracy** condition requires that the local clocks of all non-faulty nodes have a bounded drift with respect to real time.

If the communication protocol does not provide a service for clock synchronization, an appropriate solution can be realized on top of the network stack. Ramanathan et al. [RSB90] proposed two different algorithms: the convergence averaging algorithm and the convergence nonaveraging algorithm. The convergence averaging algorithm is based on a periodical exchange of the local clock values. The new value of the local clock can be set to the average or median value of all clock values [LMS85, LWL88]. In the convergence nonaveraging algorithm, each node takes over the role as system synchronizer periodically. Both algorithms require an additional round solely for the purpose of clock synchronization.

In the context of this work, this additional round can be avoided by exploiting the available knowledge about the points in time when communication is performed. Based on the expected and the actual arrival time of the individual messages, the deviation of the local clock from the logical global clock can be estimated [SS97].

The upper bound of the clock synchronization imprecision is dominated by the uncertainty of transmission delay for time-triggered communication. For event-triggered communication, the effects of potential simultaneously sent messages must be additionally considered. Another limiting factor for the clock synchronization is the clock resolution. Taking all these factors into account, a maximal synchronization skew below $200\mu s$ using Switched Ethernet and the standard real-time clock in VxWorks can be achieved in run-time systems generated with FTOS.

To tolerate f faulty clocks, Lamport et al. pointed out that the network must consist of at least n nodes, with $n > 3f$ [LMS85]. By the use of authenticated messages, the number of required nodes can be reduced to $n > 2f$ [ST87, CF94]. The used algorithm consists of two phases. During the first phase, the arrival time is attached to each incoming message. The message is discarded if the message is not within the expected time frame. The reception and storage of network messages must be performed with a high priority to guarantee a correct estimation of the arrival time. The second phase of the algorithm is invoked periodically and checks the deviation between the local and the logical global clock. For this purpose, all messages within the buffer are analyzed and the difference between the expected reception time and the actual reception time is calculated. Only messages from ECUs that take part in the clock synchronization, which is stated in the hardware model, are considered. The number of senders is counted to detect a fault of the own component. If enough valid messages are received, the local clock can be adjusted. To avoid permanent adjustments, a threshold ρ is introduced that must be exceeded before the local clock is updated. The initial temporal synchronization is based on an algorithm similar to the start-up algorithm in TTP/C [TTT03].

5.3 Fault-Tolerance Component

This section discusses the realization of the fault-tolerance mechanisms in more details. The implementation is significantly simplified by the concepts of the modeling language. This section points out the remaining subtleties. In general, the presented models allow a sophisticated and unconstrained specification of fault-tolerant

systems. The provision of a set of templates that support all possible combinations of fault-tolerance mechanisms would require much effort and would be not reasonable, as several combinations might never be used. When analyzing fault-tolerant systems, one can observe typical patterns of fault-tolerant mechanisms that are employed within one system. Examples are the use of replication with graceful degradation, if the number of faults exceeds the assumed number, or the use of rollback recovery in combination with graceful degradation. Other combinations such as using active replication in combination with rollback recovery are rare and are therefore currently not supported by existing templates. However due to the easy expandability, new templates can easily be integrated into FTOS, if required. To guarantee a maximal support of the different possible combinations, the provided templates are similar to the model split up into the four areas: error detection, error treatment, error recovery and pro-active operations. In the following, interesting details about the realization are discussed.

5.3.1 Tests

Algorithm 1: Voting - 1st Round

```
1  matrix=initializeMatrix;
2  sendResult();
3  while timeout do
4  |  //receive voting messages until timeout;
5  |  addMessage(buffer,receiveVotingMessages());
6  end
7  for all pairs of results i,j in buffer do
8  |  result=compareResults(i,j);
9  |  updateMatrix(matrix,result,i,j);
10 end
```

The first step in fault-tolerant systems is to detect errors. FTOS supports currently three different test categories: timing tests, test on port values, as well as liveliness tests. The major challenge is the consistency requirement [BSW+00]. All ECUs that execute a test must reach a consistent result to guarantee a consistent reaction. This leads to the necessity to execute consensus protocols, if a consistent decision may be compromised by faults. The exact design of the consensus protocol strongly depends on the possible faults. In this sense, the section can only provide solutions for some specific fault assumptions.

The first algorithm is proposed to realize relative tests. If no faults may affect the correct transmission of the redundant results, the implementation is straight forward and can be realized according to Equation 4.23. If however faults need to be considered, adequate consensus protocols need to be applied. In general, each ECU starts by sending the local results to the relevant ECUs. In the following, the ECUs perform consensus protocols that match the requirements stated in the unifying operation and the consistency requirement for the test result. Here, an algorithm is proposed that realizes the consensus protocols for systems with rare communication faults and for ports, where

consistent values are not necessarily required. The proposed protocol is executed in two rounds and accepts the loss of at least one message.

The first round starts with a send/receive phase and in the following an initial agreement matrix is depicted. A possible realization is depicted in Algorithm 1. The assumption for this algorithm is that every node executing the test also contributes one result. This is the case for (semi-) active replication systems.

The second round is used to compensate possible message loss. Instead of resending the port values, the participating ECUs can send the agreement matrix. This is of course only possible, if no consistent port value is required. Sending the agreement matrix can reduce the required network load and more important also increase the number of tolerated faults. The concrete number of tolerated faults depends on the concrete setting. The agreement matrices can be combined by the `OR` operation, if no message corruption and Byzantine errors are assumed. This assumption is common and used in various projects [BSW+00]. The resulting matrix can be analyzed to check the correctness of the different results. The required number ρ of agreeing results depends on the stringency used. Algorithm 2 shows an implementation for `AbsoluteMajority` or `Unanimity`.

The discussion on the different assumptions emphasizes the necessity to verify the concrete implementation with respect to the fault hypothesis. Chapter 6 points out a solution by integrating formal methods into FTOS.

Algorithm 2: Voting - 2nd round

```
1  waitForRound2();
2  sendMatrix(matrix);
3  while timeout do
4  |  m=receiveMatrix(); //receive results of 1st round
5  |  matrix=or(matrix,m);
6  end
7  for each line i in matrix do
8  |  sum=sumUpLine(i); //check correctness of i
9  |  if sum< ρ then
10 |  |  notifyError(i);
11 |  end
12 end
```

The implementation of liveness tests must also consider communication faults for a correct implementation. For a reliable network, the system can use system application messages instead of additional life sign messages to signal the availability to the monitoring nodes. The network component stores the time when the last message was sent to a monitoring node in an auxiliary variable. When the system reaches a program counter with a liveness test, the system simply checks, whether at least one message was sent during the evaluation period to all monitoring ECUs. If no message was sent to one monitoring node, an additional life sign is sent. If on the other hand an unreliable network is used, each ECU must be sure that all monitoring ECUs received at least one message within the past evaluation interval. Therefore, the use of an additional

life sign including a consensus phase is usually required. The concrete implementation depends again on the fault hypothesis.

5.3.2 Error Treatment

The system can react to detected errors in three different ways: switching to a correct replica (redundancy management), recover from the effects of a transient fault by restoring a previous state and repeating the execution (rollback recovery), or switching into a degraded mode by changing the program counter and/or the values of some variables of the system. All these mechanisms can be implemented trivially in FTOS, due to the time-triggered nature, the available previous knowledge, and the test consistency. In the following, the most important aspects are briefly highlighted.

Redundancy Management

Using hardware or software redundancy in the form of passive, semi-active or active replication is the main concept used in hard real-time systems. Especially (semi-) active replication provides possibilities to mask errors without taking time delays into account. Switching to a correct replica (master selection) in passive or semi-active replication can be implemented easily using the trigger concept. In general, the problems in the context of redundancy management are the realization of state synchronization in cold-standby systems, the timely recognition of failures, the consolidation of the output, and the handling of event-based inputs. The first two problems are solved by the concepts of the model: the separation of states and the time-triggered execution.

The purpose of output consolidation is to map the replicated logical outputs of each channel onto the actual physical outputs, in such a way that the latter is in an error free or in a safe position [PABD+99]. The output consolidation is the ultimate error confinement barrier. All components after the output consolidation form of course a single-point of failure and must therefore be designed carefully. It is inevitable that the output technique has to be realized in an application-specific manner. Therefore, most of the research projects leave this problem to the developer. In contrast, FTOS allows a flexible output for replicated systems. By selecting the replication type, the developer can specify, whether all correct or only one correct replica should perform the output. The test consistency guarantees a consistent selection of the acting replicas. Failures of these replicas after the decision can be detected by additional tests. These tests can easily be integrated, as well as appropriate reactions. Of course, this approach assumes fail-silent behavior of the hardware executing the run-time system or the toleration of temporarily wrong outputs by the controlled system.

Another problem is the handling of event-based inputs in (semi-)active replication. If the event occurs, while the ECUs are executing the input actors, some ECUs might observe the event, whereas others might observe the event not until the next round. The standard solution is to execute a consensus phase for the input results. However, this additional consensus phase requires time that might not be available. FTOS solves this problem by the `IGNORE` reaction for fault recovery. The minority of both groups is assumed to be erroneous and therefore cannot perform output operations. However,

the affected ECUs continue their operation and the system states will converge in the following rounds. By the appropriate use of error counters, real errors are treated in a correct way. However, this approach is only valid if no mode changes are triggered by the incoming events that might lead to a partition of the system into two subsystems with different schedules.

Rollback Recovery

In case of a transient or intermittent fault, a repeated execution of some function can overcome the error [RLT78]. This strategy is called rollback recovery. Instead of executing the same function again, it can also be promising to use some other function or to execute the function on another processing unit. In the model, this approach can be realized by using passive replication in combination with checkpoints.

Rollback recovery is a rather simple approach; the only difficulties arise in designing and generating the checkpoints. However, the creation of checkpoints becomes trivial in FTOS due to the separation of state and functionality. Another common problem of checkpointing is the domino effect [Ran75]. In asynchronous systems, each unit creates checkpoints according to its own schedule. If one unit discovers an error and performs a rollback operation, orphan or missing messages can be a result. To treat this problem, the other units must also perform rollback operations, which can lead to a domino effect. One obvious solution is the creation of a global checkpoint and a synchronously performed rollback operation [BHMR95]. The realization of such a mechanism in a time-triggered environment becomes trivial.

5.3.3 Error Recovery

The concrete mechanisms for the recovery of an erroneous unit are typically application-dependent. Therefore, only the very basic mechanisms such as restart, reboot or fail-stop are offered. An interesting point about error recovery is integration. For integration, a successful state synchronization is the prerequisite. The state synchronization covers the synchronization with the logical time and the transfer of the system state in form of the values of related ports, trigger decisions, and test results.

FTOS supports both silent and non-silent integration. In silent integration, the running system is not directly influenced by the integrating unit. The ECUs of the running system simply check whether an integration request is pending and forward in case of a pending request the required state information. The integration is done by the integrating unit itself and is realized logically during the subsequent tests that will eventually lead to a full integration. To guarantee an enduring success of the integration, no relevant tasks are allowed to be executed on the running ECUs at the time of integration. Otherwise, the update of the port values at the end of the task execution could again lead to an exclusion of the previously integrated unit. This condition can be checked during model validation by analyzing the time schedule. A possibility to guarantee the correctness of the received data is to perform voting on the transmitted data, if ECUs are forwarding redundant state information to the integration unit.

Non-silent integration is performed, if the system state is affected directly by the in-

tegration. Reasons can be the continuation of the system with a different program counter or the update of test results / port values. The implementation can be done analog to the state restoration concepts in GUARD [BCGG97, BGG+98]. The integrating unit sends an integration request and waits for state synchronization messages (**receive state**). When an integration request is observed and the running system reaches an integration point, the active units enter the **send state**. The integrating unit analyzes the receive messages, perform tests to verify the consistency of the received state and signals the running system the successful completion of the integration. The execution of the system is continued at the specified program counter or, if no program counter was specified, at the current position.

5.4 Fault-Tolerant Scheduling

The scheduling component is a good example to outline the potential of optimization with respect to the execution model. Bondavalli pointed out that design environments are missing, which allow the designers to evaluate the impact of different priority schemes and scheduling algorithms [BFLS01]. The philosophy of FTOS is contrary to this requirement in the sense that FTOS automates the appropriate selection of scheduling parameters, as it was requested by Stankovic [Sta88]. The concept of logical execution times enables this approach. Low-level implementation details are abstracted and the concrete design of the scheduling algorithm can be realized by the code generator.

This section discusses the implementation of scheduling algorithms that are potential candidates for FTOS. The discussed algorithms comprise Earliest-Deadline First (EDF) [LL73], Stack-based (SB) [TTT05], Fixed Priority (FP) scheduling, as well as Cyclic Executive (CE) [BS88]. Without loss of generality, the section assumes that the algorithms are implemented on top of a standard priority-based scheduler. For each algorithm, an implementation is discussed that exploits the characteristics of the execution model. The assumptions are the following:

A1 The application consists of a fixed number of tasks.

A2 There are no precedence constraints and shared resources.

A3 For all tasks, the start time and deadline is known in advance.

A4 Worst case execution time (WCET) assumptions may be available.

The section points out that all algorithms can be implemented with constant run-time O(1) for each scheduling decision independent from the number of tasks. All algorithms, except Fixed Priority, can be implemented in a way that at most three priority levels are required. Therefore, the algorithms can be executed also on platforms with an operating system providing only a limited number of priority levels [SKS95]. The algorithms are evaluated using several criteria with a focus on robustness and performance. In the context of the research project TDL, a similar evaluation was performed [Far06]. However, there was neither a discussion about the run-time overhead, nor a consideration of WCET exceedance. At the end of this section, a scheduling algorithm is presented that combines the advantages of EDF and CE.

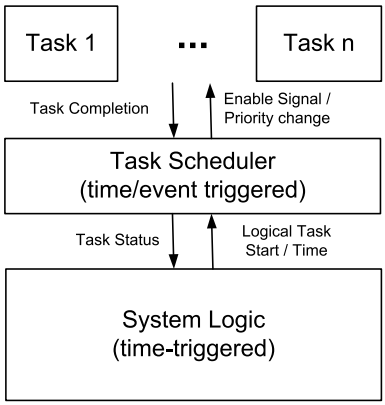


Figure 5.2: Component Interaction for Scheduling

5.4.1 Evaluation of Standard Scheduling Algorithms

This section discusses the implementation of four scheduling algorithms in the context of the simple task model: Cyclic Executive (CE), Stack-Based (SB), Earliest-Deadline First (EDF) and Fixed Priority (FP) scheduling. In all cases, preemptive scheduling is used since non-preemptive scheduling is not adequate for control systems with tasks having different periods. The main idea used to implement the algorithms on top of a priority-based scheduler is depicted in Figure 5.2. The task of the scheduling component is to select the appropriate task that should be currently executed on the CPU. For the FP algorithm, no scheduling component is required. The scheduling component can initiate the execution of a task by unblocking a semaphore, whereas no possibility exists to stop a task during the execution of the task function. However, it is possible to suspend the execution by manipulating the priorities. To realize the suspension, the availability of two priority levels used for the scheduling is assumed: one priority level (high) to execute the selected task and another priority level (low) to suspend previously started tasks. By the use of a notification service that is again implemented by a semaphore, the task can signal its completion to the scheduling component. If the scheduling component does not contain a list concerning the task timing, which is the case in EDF scheduling, the system logic signals the logical start time of a task to the scheduling component. Otherwise, only time signals are required. Vice versa, the system logic is able to retrieve the states of the tasks including information about deadline violations.

A simple example is used for the illustration of the different algorithms: a system consisting of two tasks $\mathtt{t}_1, \mathtt{t}_2$ that are executed periodically. The logical execution time is depicted in Figure 5.3. The duration of the mode cycle is $\mathtt{30ms}$, and the worst case execution times are $\mathtt{WCET}(\mathtt{t}_1) = \mathtt{16ms}$ and $\mathtt{WCET}(\mathtt{t}_2) = \mathtt{4ms}$. The actual execution is assumed to be $\mathtt{15ms}$ and $\mathtt{3ms}$ for task $\mathtt{t}_1$ respectively $\mathtt{t}_2$.

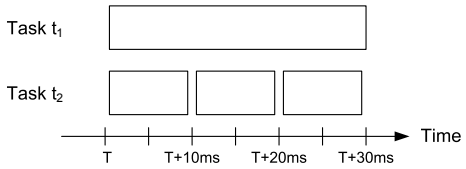


Figure 5.3: Scheduling Example - Logical Execution Times

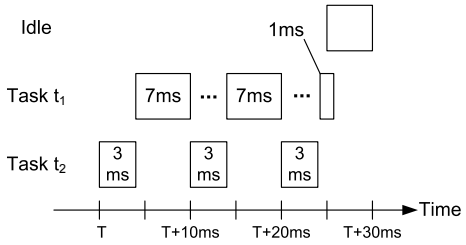


Figure 5.4: Actual Execution using Fixed Priority Scheduling

Fixed Priority

The main advantage of Fixed Priority scheduling is its simplicity. A priority is assigned to each task during code generation. To guarantee a correct assignment, information about the WCET is required. In addition, the priority assignment is only possible if enough priority levels are available.

The realization of Rate-Monotonic is straightforward. Since all task priorities can be calculated offline, there is no need for a scheduling component. Thus, the start of the task is directly performed by the system logic, e.g. by releasing a semaphore. By using a list containing all finished tasks, the system logic can check, whether a task could complete its execution on time. The execution in the context of the example is depicted in Figure 5.4. The assignment of the priorities can be achieved in this example according to the Rate Monotonic [LL73] method: the task with the shortest period, here $\mathtt{t}_2$, receives the highest priority. Therefore, $\mathtt{t}_2$ starts the computation at time $\mathtt{T}$ and interrupts the execution of task $\mathtt{t}_1$ at the points in time $\mathtt{T} + \mathtt{10ms}$ and $\mathtt{T} + \mathtt{20ms}$.

Earliest-Deadline-First Scheduling

Recently, the advantages of Earliest-Deadline-First (EDF) scheduling introduced by Liu [LL73] in comparison to Fixed Priority were outlined by Buttazzo [But05]. The main advantage of preemptive EDF is its optimality: a correct schedule is always found, if one exists. The disadvantage is the implementation complexity, in case the scheduler is developed on top of a kernel using a set of fixed priority levels. Buttazzo stated that the implementation *is not easy, nor efficient*. The main problems identified were the necessity to remap priorities if not enough priority levels are available and the inefficient management of the deadline queue.

In the context of FTOS, these problems can be easily avoided. Because synchronization points are excluded by design, the system can select the task with the earliest deadline

and execute this task without having to consider that the task could block. Therefore, the system does not have to assign priorities for all tasks. The deadline management is mitigated by the existence of discrete deadlines. Since all start times and deadlines are known in advance and the number of deadlines is bounded, the run-time complexity can be decreased significantly. It is actually possible to realize the deadline management by data structures that allow the insertion of new tasks and the extraction of the task with the earliest deadline in constant time O(1).

The EDF scheduling algorithm can be realized as follows: instead of assigning priorities to all ready tasks $t \in T_{ready}$ based on their deadlines and starting all tasks, the proposed algorithm starts the task t_e with the earliest deadline and assigns the priority high. All other tasks remain blocked. If a new task t_{new} with an earlier deadline becomes ready during the execution of the task t_e, the priority low is assigned to the currently running task t_e. By assigning the high priority to t_{new}, the new task is now executed by the priority-based scheduler. This procedure is repeated every time a new task with an earlier deadline than the currently executed task becomes ready. If a task finishes its execution, the task with the next deadline is unblocked, if necessary, and executed with the priority high.

The implementation is depicted in Algorithm 3. The scheduler is invoked with an empty set of ready tasks and executes a while loop waiting for some external event: the incoming of a new task or the task completion. In case of a new task, the task is inserted into the set of ready tasks and the task is marked as not started. If the deadline of the new task is earlier than the deadline of the currently executed task, the latter is suspended by assigning the low priority and the new task is started. Otherwise, the scheduler does nothing. The event of a task completion triggers the removal of this task and the selection of the task with the next deadline from the set of ready tasks. The priority of the next task is set to the high priority and the task is unblocked, if necessary.

For simplicity reasons, the exact data structure used for deadline management of the ready tasks is not specified. The simplest solution is the use of a sorted list, but this leads to an insertion complexity of $O(n)$. Since the number of deadlines is limited, it is much more efficient to use queues for the different deadlines and hash tables instead of a sorted list. This reduces the complexity to $O(1)$. In the example, task t_2 starts the execution because of the earlier deadline in comparison with task t_1. The CPU is assigned to task t_1 after the completion of the first task execution of t_2. At time $T + 10ms$, task t_2 interrupts task t_1. This is not the case at time $T + 20ms$ because now both tasks have the same deadline. The actual execution is depicted in Figure 5.5.

Stack-Based Scheduling

The time-triggered operating system TTPOS [TTT05] uses stack-based scheduling as scheduling strategy. It is important to mention that the used notion of stack-based scheduling contradicts the more general notion of Baker [Bak91]. For stack-based scheduling, a task definition list (TADL) defines the points in time, when a task is started. The term **stack-based scheduling** is derived from the algorithm strategy: all tasks that were started, but could not yet complete, are stored within a stack. Only

Algorithm 3: Earliest-Deadline First Scheduler

```
1  T_ready := newEDFQueue();
2  Started := newTaskStatusArray();
3  t_e := ⊥;
4  t_new := ⊥;
5  e := ⊥;
6  while true do
7      e := waitForEvent();
8      if (e.isNewTask()) then
9          t_new := getTask(e);
10         Started[t_new] := false;
11         T_ready.insert(t_new);
12         if (t_e.deadline > t_new.deadline) then
13             setPrio(t_e, low);
14             t_e := t_new;
15             setPrio(t_e, high);
16             unblock(t_e);
17             Started[t_e] := true;
18         end
19     else t_e completed the execution
20         T_ready.remove(t_e);
21         if (T_ready = ∅) then
22             t_e := ⊥;
23         else
24             t_e := T_ready.getMin();
25             setPrio(t_e, high);
26             if (!Started[t_e] then
27                 unblock(t_e);
28                 Started[t_e] := true;
29             end
30         end
31
32 end
```

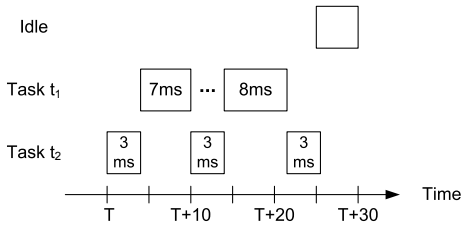


Figure 5.5: Actual Execution using Earliest-Deadline-First Scheduling

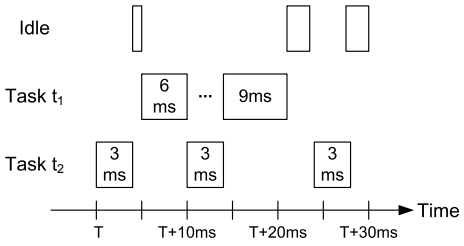


Figure 5.6: Actual Execution using Stack-Based Scheduling

the task on top of the stack is executed, all other tasks are suspended. A new task is always inserted on top of the stack. After task completion, the task is removed from the stack. The task definition list can be calculated during code generation, but information about the WCET is required [FM98]. The idea is to simulate the EDF execution using the WCET assumptions.

The design of this scheduling algorithm on top of a priority-driven scheduler of a common operating system is easy: the scheduler simply waits for the next start time or the completion of the task currently running. If a new task starts, the scheduler assigns the priority `high` to the new task, unblocks the task and sets the priority of the currently executed task to `low`. If a task is completed, the priority of the previously executed task is raised to `high`. The management of the stack is trivial.

The actual execution is depicted in Figure 5.6. The according TADL has the following entries:

$T + 0:$ t_2
$T + 4:$ t_1
$T + 10:$ t_2
$T + 24:$ t_2

Cyclic Executive Scheduling

Cyclic Executive [BS88] is a time-triggered scheduling algorithm. The execution time is partitioned into slots, which are assigned to the tasks. The main advantages are the low online overhead and the guaranteed execution time for each task. The realization of CE is similar to the Stack Based solution. The execution plan can be calculated by simulating an EDF execution using the WCET assumptions. In addition, the results can be remanufactured to avoid unnecessary context switches. The result of the generation process is a list containing all start and continuation times of the tasks chronologically ordered. The online execution is similar to Stack-Based scheduling with the exception that no dynamic stack is needed at run-time. The actual execution in the example is depicted in Figure 5.7.

5.4.2 Evaluation

This section evaluates the proposed algorithms with respect to implementation complexity, run-time overhead and robustness.

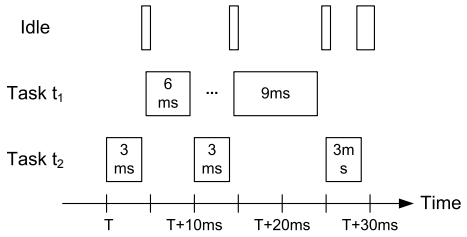


Figure 5.7: Actual Execution using Cyclic Executive Scheduling

Task Number	5	10	20	50	100
EDF	6,0	5,9	5,4	5,3	5,9
FP	2,2	2,1	1,8	1,6	2,2
CE	5,6	5,6	5,7	5,8	6,6
SB	5,7	5,7	5,8	5,9	6,7

Table 5.1: Average Time for a Scheduler Decision in μs

Implementation Complexity and Run-Time Overhead: All algorithms can easily be implemented. Fixed Priority requires no scheduling component at all. The implementation of EDF is a little bit more complex, but easily manageable. While the implementation of SB and CE is simpler, these algorithms require tools to generate the execution plan. Since this plan generation is based on the simulation using EDF, none of the algorithms SB, CE, or EDF is superior.

The same result can be observed with respect to the run-time overhead. Table 5.1 shows the experimental results: the average time to schedule one task (starting and stopping a task) including the code of the system (not included in CE and SB) and scheduling component (not included in FP). For the benchmarks, a VxWorks 6.3 operating system running on an Intel Pentium IV (1,6 GHz) processor was used. Different task sets were applied to test the behavior of the algorithms. The benchmarks show the advantages of FP in comparison to the other algorithms. It is of course the fastest algorithm, since no scheduling component is required. The decrease in the average time for a scheduling decision for FP and EDF can be explained by the declining influence of the execution time spent for the system component in comparison to the overall spent execution time. This explains also the difference in comparison to SB and CE, since for these benchmarks no system component was included. The reason, why SB and CE consume a similar amount of time in comparison to EDF despite the lower implementation complexity, is the different implementation of the operating system calls. VxWorks implements a semaphore operation much faster than the wait operation for a timed signal. Operating system internals are also responsible for the increase of the average time for 100 tasks for all algorithms.

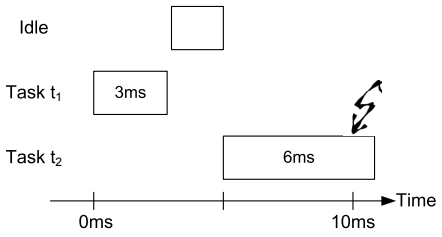


Figure 5.8: Scheduling Failure

Robustness

One main requirement in fault-tolerant systems is robustness. The system should tolerate WCET exceedance if possible. Due to the optimality of EDF, it will always find a schedule, if one exists. This is not true for FP, SB or CE. Figure 5.8 points out one possible scenario where CE and SB fail. The plan consists of two tasks t_1,t_2 with WCET 4ms and a deadline $d_1 = d_2 =$ 10ms. A possible execution plan is to split up the available execution time into two slots of length 5ms. If the actual execution times are $e_1 =$ 3ms and $e_2 =$ 6ms, both algorithms fail to find a valid schedule, although a valid execution plan exists. A similar example can be constructed for FP.

Another aspect of robustness is that correct tasks should not be negatively influenced by erroneous tasks. It is obvious that plan-based scheduling is the best solution to solve this problem. Since every task has its own execution slots, no task can influence the other tasks. This is not true for EDF [But05], RM, and stack-based scheduling.

Summary of Evaluation

To conclude this section, the evaluation results are summarized. Interestingly, SB, which is currently used in time-triggered systems, turned out to be the worst choice. Concerning the implementation complexity and the benchmark results, the ideal algorithm is FP. Its implementation is trivial, and due to the fixed priorities, the run-time system overhead is minimized. Disadvantages of FP are the need of WCET information and the lack of robustness. EDF can be implemented without WCET information and without compromising the system performance. Due to its optimality, EDF is an ideal solution if no information about WCET is available. If information about WCET is available and erroneous tasks are assumed to exceed the scheduled time, CE should be preferred. Recapitulating, a combination of plan-based scheduling and EDF would be the best solution: EDF with guaranteed time slots. This solution is discussed in the next section.

5.4.3 Earliest-Deadline-First with Guaranteed Time Slots

In the previous section, different scheduling algorithms were discussed and the advantages of both Earliest-Deadline-First (EDF) and Cyclic Executive (CE) were identified. While EDF allows a toleration of WCET violations, if a possible schedule is available,

other tasks might be affected by an erroneous task. In contrast, CE isolates the different tasks and avoids any influence, but is inflexible if some time is left that can be used by the exceeding task to complete its execution. The problem resembles the requirement to schedule periodic hard real-time tasks and aperiodic soft real-time tasks. In FTOS, the additional time that is required by a task exceeding its WCET can be handled similar to the time required by an aperiodic task. The literature proposes two different solutions for this problem: background scheduling [Liu00] or slack stealing algorithms [CC89, LRT92]. The background scheduling approach executes the erroneous tasks, whenever no other task is scheduled. However, this approach may delay the execution and lead to an unnecessary violation of a deadline. A more promising approach is the use of slack stealing algorithms. The term **slack** denotes the maximum time interval for which the remaining slots can be postponed without violating any deadline. Slots can also be shifted forward, if one periodic task completes its execution earlier than expected. The obtained slack can be used later on for tasks exceeding their WCET. In summary, the perfect solution is to use flexible slots that can be shifted to front to reclaim unused processor time or shifted backwards, if a task exceeds its WCET and some slack is left. This approach resembles the concept of slot shifting [Foh95, IF99].

To realize this approach, the run-time system has to efficiently keep track of the remaining slack. Tia [Tia95] proposed an algorithm to reduce the run-time overhead by using a pre-computed slack table. However, this algorithm requires an overhead of O(n) to calculate the current slack for a system with $\mathtt{n}$ tasks.

This thesis points out that the complexity can be further reduced to $\mathtt{O}(1)$ by continuously updating the slack. Without loss of generality, each task is assumed to be executed exactly once within a schedule. In addition, the tasks are assumed to be ordered by their deadline. The initial slack $\sigma_{\mathtt{i}}(0)$ for task $\mathtt{t_i}$ can be calculated and precomputed by the following equation [Tia95]:

$$\sigma_i(0) = d_i - \sum_{d_k \leq d_i} e_k \tag{5.1}$$

In the equation above, $\mathtt{d_i}$ refers to the deadline of $\mathtt{t_i}$ and $\mathtt{e_i}$ specifies its assumed (worst-case) execution time. For an interval $[\mathtt{d_j}, \mathtt{d_k}]$, the initial slack can be denoted by $\omega(\mathtt{j};\mathtt{k}) = \mathtt{min_{j \leq i \leq k}}(\sigma_{\mathtt{i}}(0))$.

Figure 5.9 shows an example that is used in the following as running example. The schedule consists of five tasks. The start time $\mathtt{s_i}$, execution time $\mathtt{e_i}$, deadline $\mathtt{d_i}$, and initial slack is specified in Table 5.2.

Based on the initial slack, the current slack can be computed: every time a task is executed with a later deadline, the processor idles, or a task exceeds its scheduled time, the slack decreases. In general, the following information must be available to compute the current slack for each task:

1. **Total processor idle time** $\mathtt{I}$: The idle time of the processor since the start of the execution plan.
2. **Overrun** $\mathtt{O}$: The extra time spend for tasks exceeding their WCET assumptions
3. **Gain** $\mathtt{g_i}$: The unused processor time by task $\mathtt{t_i}$
4. **Execution Time** $\xi_{\mathtt{i}}$: The completed portion of each task $\mathtt{t_i}$

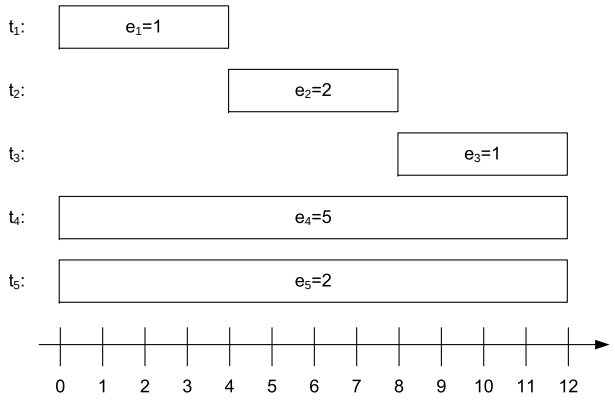


Figure 5.9: Example used for Slack Calculation

Task	s_i	e_i	d_i	σ_i
t_1	0	1	4	3
t_2	4	2	8	5
t_3	8	1	12	1
t_4	0	5	12	1
t_5	0	2	12	1

Table 5.2: Start Times, Execution Times and Deadlines in Example Used for Slack Calculation

At time τ, the slack of each task $\mathtt{t_i}$ with $\mathtt{d_i} > \tau$ can be calculated by the following equation:

$$\sigma_i(\tau) = \sigma_i(0) + \sum_{d_k \leq d_i} g_k - I - O - \sum_{d_k > d_i} \xi_k \tag{5.2}$$

All unused execution time of tasks that have a deadline not later than the deadline of task $\mathtt{t_i}$ is added. The total idle time and the total spend overload is subtracted, as well as the sum of all processor time spend for tasks with a later deadline. The current slack of the system is then again the minimum of all slacks. This brute force computation has a complexity of O(n).

The main idea to reduce this complexity is to calculate the slack for intervals instead for each task. The intervals are defined by the start times and deadlines of the tasks. In the running example, the relevant intervals are $[0, 4]$, $[4, 8]$ and $[8, 12]$. At the boundary of an interval τ, the preceding interval includes all deadlines $\mathtt{d_i} = \tau$, while the subsequent interval includes the start times $\mathtt{s_j} = \tau$. To guarantee a correct scheduling, all tasks with a deadline equal to the end of the interval must get the scheduled processor time that is specified by their WCET assumption. For a specific interval $\mathtt{I}[\tau, \tau']$, $\mathtt{E_I}$ refers to the execution time of all tasks with a deadline equal to τ'.

$$E_{I_{[\tau,\tau']}} = \sum_{t \in Task : d_t = \tau'} e_t \tag{5.3}$$

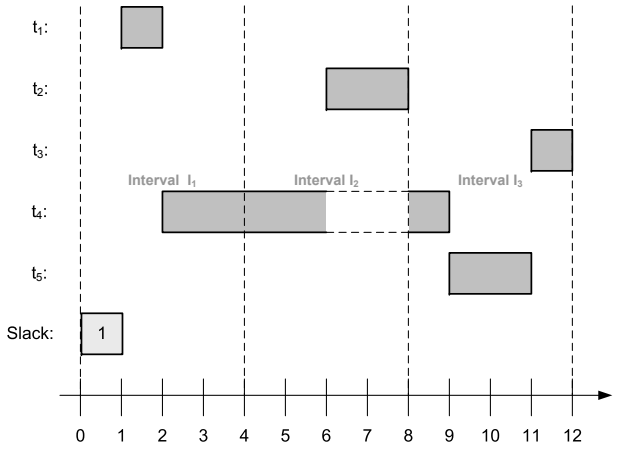


Figure 5.10: Worst Case Execution Plan

In addition, some execution time must be spend for tasks with a later deadline, if subsequent intervals do not provide enough processor time to schedule all tasks with a relevant deadline. This additional time $\mathtt{L_I}$ can be derived from the following equation:

$$L_{I_{[\tau,\tau']}} = \begin{cases} 0, & \tau' - \tau - E_{I_{[\tau,\tau']}}) < 0 \\ min\big(\tau' - \tau - E_{I_{[\tau,\tau']}}, \sum_{I_{[t,t']}:\tau'<t'}(E_{I_{[t,t']}} + S_{I_{[t,t']}} - (t'-t))\big), & otherwise \end{cases} \tag{5.4}$$

The interval $\mathtt{I}_{[\tau,\tau']}$ can not execute any later task, if the required execution time $\mathtt{E}_{\mathtt{I}_{[\tau,\tau']}}$ exceeds the available execution time $\tau - \tau'$. If otherwise some execution time is left, $\mathtt{L_I}$ is the minimum of the available and the required time. The remaining processor time in each interval is the slack $\mathtt{S_I}$ that can be used for tasks exceeding their WCET assumption:

$$S_{I_{[\tau,\tau']}} = min(0, \tau' - \tau - E_{I_{[\tau,\tau']}} - L_{I_{[\tau,\tau']}}) \tag{5.5}$$

The above listed equations are only valid, if a valid execution plan exists. A graphical representation illustrates the meaning of the equations. The required time for scheduled tasks within one interval and the resulting initial slack time of each interval can be derived from the worst case execution plan. This plan refers to an execution plan that depletes all available slack immediately. The worst case execution plan for the running example is depicted in Figure 5.10. The required processor time within the intervals is 1 for $\mathtt{I}_1$, 2 for $\mathtt{I}_2$, and 8 for $\mathtt{I}_3$. As each interval is only 4 time units long, it is necessary to execute a task with a deadline later than 4 in interval $\mathtt{I}_1$ for two time units and similarly a task with a deadline later than 8 in interval $\mathtt{I}_2$ for two time units. The initial slack of interval $\mathtt{I}_1$ is 1, for the other intervals zero. The slack time can be used for tasks exceeding their WCET. If no such task is available, it can be used to execute other started tasks in a pro-active way to increase the slack of subsequent intervals. The slack time of the current interval decreases, if additional tasks are executed in a pro-active way, the processor is assigned to a task that exceeded its WCET assumption, or if the processor is idle.

The computation of the initial slack is straightforward starting with the precomputed execution plan based on the assumed WCET. The worst case execution plan can be derived starting from the end of the execution plan. For each interval all time slots are shifted to the back. The remaining free time slots are filled by shifting time slots from other intervals into this slot under consideration of the start times. If no time slot is found for the free slot, a slack time slot is created. Due to the deadline driven approach, the order of each item is not important.

The calculation of the slack time for the intervals can be performed in $O(1)$ at run-time. For a scheduling decision only the slack time of the current interval is required instead of having to determine the minimum of the slacks of all tasks.

Another difference results from the different goals of integrating aperiodic tasks and tolerating WCET violations. While response time is the major goal when integrating aperiodic tasks, the goal in tolerating WCET violations is the avoidance of deadline violations. Therefore, the assignment of additional time for exceeding tasks can be deferred until no other task with a deadline earlier or equal to the deadline of the exceeding task is available. This helps to minimize context switches.

Algorithm

The concrete algorithm is similar to the presented EDF algorithm. Each task is managed in a struct (t, e_t, a_t, i_t). The difference is that besides deadlines, the scheduled execution time e_t, the already spent processor time a_t, and the relevant interval i_t has to be available for each task.

In addition to the standard queue T_{edf} containing the tasks that need to be scheduled, an overrun queue T_o is used to store all tasks that have exceeded their WCET assumptions and could not complete their execution. The management of this queue is similar to the queue used for standard tasks and is ordered by deadlines.

Information about each interval i is contained in variables E_I, L_I, S_I, and A_I. The latter specifies the time that was already assigned to a task with a deadline in I. The currently executed interval is specified in variable i, a pointer t_c refers to the currently executed task. Finally, variable τ denotes the time, when the execution of the currently executed task was started and $time$ refers to the current time. Variable $extra$ is used to sum up the processor time spent for pro-active execution, as this time increases the slack of subsequent intervals. In fact, the slack time variables S_I are not required for this algorithm, but used for illustration purpose. The initialization of the algorithm is described in Algorithm 4. After the initialization, the algorithm performs an infinite loop that executes in four phases:

1. Execution of tasks with a deadline equal to the interval end: During this execution, the tasks are executed according to the scheduled time. Tasks exceeding their WCET assumption are put into the overrun queue T_o. If a task releases the processor earlier than expected, the slack increases by the amount of unspent time. The code of the first phase is specified in Algorithm 5.
2. Execution of tasks with a deadline later to complete L_I: This phase is similar to the first phase - if a task releases the processor time earlier than expected, the slack

Algorithm 4: EDF with Guaranteed Execution Times - Initialization

```
1  T_edf := newEDFQueue();
2  T_o := newEDFQueue();
3  t_c := ⊥;
4  v := ⊥;
5  τ := ⊥;
6  extra := 0;
7  i := getFirstInterval(T_edf);
8  newInterval := false;
9  forall j ∈ Intervals do
10 |  L_j := getPrecomputedExectionTime(j);
11 |  L_j := getPrecomputedExectionTimeForLaterIntervals(j);
12 |  S_j := getPrecomputedSlack(j);
13 |  A_j:=0;
14 end
```

time of the current interval increases by the unspent time. Tasks exceeding their WCET assumptions are put into the overrun queue. In addition, the variables A_i have to be updated (see Algorithm 6).

3. Proactive-execution of tasks or assignment of additional processor time to tasks in T_o: The scheduler selects the task with the earliest deadline in the queues T_{edf} and T_o and executes this task. To minimize the number of context switches, the scheduler prefers tasks within the queue T_{edf}. The slack of the current interval decreases during the execution of the third phase. Variable `extra` is used to sum up the additional time spent for pro-active operation. For tasks of the queue T_{edf}, the relevant variables A_I and a_t have to be updated, unless the task violates the WCET assumption. The code is specified in Algorithm 7.

Algorithm 5: EDF with Guaranteed Execution Times - 1st phase

```
1  t_c := T_edf.getMin();
2  while t_c.i_t = i do 1st phase
3  |  setPrio(t_c.t, high);
4  |  τ := time;
5  |  ret := timedWait(Sem_schedule, t_c.e_t − t_c.a_t);
6  |  setPrio(t_c.t, low);
7  |  T_edf.remove(t_c);
8  |  if !ret then task exceeded WCET
9  |  |  T_o.insert(t_c);
10 |  else update slack if task finished computation earlier
11 |  |  S_i := S_i + (t_c.e_t − t_c.a_t) − (time − τ);
12 |  t_c := T_edf.getMin();
13 end
```

Algorithm 6: EDF with Guaranteed Execution Times - 2nd phase

```
1  while L_i > 0 do 2nd phase
2      setPrio(t_c.t, high);
3      τ := time;
4      if L_i > t_c.e_t − t_c.a_t then task can be finished before expiration of L_i
5          ret := timedWait(Sem_schedule, t_c.e_t − t_c.a_t);
6          setPrio(t_c.t, low);
7          T_edf.remove(t_c);
8          A_pi := A_pi + t_c.e_t − t_c.a_t;
9          if !ret then task exceeded WCET
10             T_o.insert(t_c);
11         else update slack if task finished computation earlier
12             S_i := S_i + (t_c.e_t − t_c.a_t) − (time − τ);
13         L_i := L_i − (t_c.e_t − t_c.a_t);
14     else
15         ret := timedWait(Sem_schedule, L_i);      // execute t_c for time L_i
16         setPrio(t_c.t, low);
17         if !ret then task did not finish
18             A_pi := A_pi + L_i;
19             t_c.a_t := t_c.a_t + L_i;
20             L_i := 0;
21         else task finished computation earlier
22             T_edf.remove(t_c);
23             L_i := L_i − (t_c.e_t − t_c.a_t);
24             S_i := S_i + (t_c.e_t − t_c.a_t) − (time − τ);
25             A_pi := A_pi + (t_c.e_t − t_c.a_t);
26
27     end
28     t_c := T_edf.getMin();
29 end
```

4. Preparation phase for next interval: at the end of an interval, the variables concerning the current interval are reset. The interval is changed to the next interval and the slack time, the additional processing time L, and variable extra are updated. The new slack time results from the available processing time (time span between begin and end of the interval), the remaining time to complete tasks with a deadline equal to the end of this schedule and the required time for the execution of future tasks. The concrete operations are listed in Algorithm 8.

For reasons of simplicity, the algorithm assumes that the whole execution of the schedule component is performed in zero time. Therefore, the execution of a task can only be interrupted during the third phase. Deadline violations are detected and handled by analyzing the state of the task.

Summary: Within this section, different scheduling algorithms were evaluated for the

Algorithm 7: EDF with Guaranteed Execution Times - 3rd phase

```
1  while true do 3rd phase: will be aborted by system logic
2      v := T_o.getMin();
3      t_c := T_edf.get();
4      τ := time;
5      if (t_c = ⊥ ∨ (v! = ⊥ ∧ v.d_t < t_c.d_t)) then execute exceeding task
6          setPrio(v.t, high);
7          ret := wait(Sem_schedule);
8          setPrio(v.t, low);
9          S_i := S_i − (time − τ);
10         if !newInterval then task completed
11             T_o.remove(t_c);
12         end
13     else if t_c! = ⊥ then perform pro-active execution of tasks in T_edf
14         setPrio(t_c.t, high);
15         ret := timedWait(Sem_schedule, t_c.e_t − t_c.a_t);
16         setPrio(t_c.t, low);
17         S_i := S_i − (time − τ);
18         if newInterval then signal to start next interval
19             A_pi := A_pi + time − τ;
20             t_c.a_t := t_c.a_t + time − τ;
21             extra := extra + time − τ;
22         else
23             T_edf.remove(t_c);
24             A_pi := A_pi + t_c.e_t − t_c.a_t;
25             extra := extra + t_c.e_t − t_c.a_t;
26             if !ret then execution time expired
27                 T_o.insert(t_c);
28             end
29         end
30     else no task contained in both queues
31         Wait(Sem_schedule) ;                    // wait for next interval
32         S_i := S_i − (time − τ);
33     if newInterval then
34         newInterval := false;
35         break;                                  // complete round
36     end
37 end
```

Algorithm 8: EDF with Guaranteed Execution Times - Preparation of Next Interval

```
1  L_i := getPrecomputedExectionTimeForLaterIntervals(i);
2  S_i := getPrecomputedSlack(i, T_edf);
3  A_i:=0;
4  i := getNextInterval(i);
5  if i = getFirstInterval() then
6  |   extra := 0;
7  else
8  |   extra := extra − A_i;
9  |   L_i := max(0, L_i − extra);
10 |   S_i := i.end − i.begin − ((E_i − A_i) + L_i);
11 end
```

use in FTOS. As a result, three different algorithms seem to be most adequate. If a scheduling plan for fixed priorities exists and no WCET violations are expected, rate monotonic is a perfect solution. If no such plan exists or no WCET information is available, Earliest-Deadline-First should be applied. If WCET information is available, the perfect solution is to use an algorithm based on EDF with guaranteed execution times. This algorithm combines both the tolerance of WCET violations of EDF and the fault containment of CE.

5.5 Evaluation

This section discusses the available prototypes and outcomes of the demonstrator applications. Templates for two different run-time systems are available. The first run-time system is a generic, POSIX [COR94] compatible run-time system that requires the availability of a real-time operating system. The used operating system is currently VxWorks 6.3 [1]. The second run-time system is designed especially for constrained platforms. The targeted hardware platform is an Atmel AT90CAN128 processor [Atm07]. Since no operating system is used, the run-time system is dependent of the hardware, but can be highly optimized.

The run-time systems are generated tailored for the specific application to reduce the overhead penalties of the middleware approach. A good example for the optimization is the application-specific selection of an optimal scheduler. In comparison to other middleware approaches, the overhead is significantly decreased. Unfortunately, there are not many solutions available that can be used for a direct comparison. The overhead of the middleware used in HTL is estimated by 60 to 300μs per time instant. In contrast, the overhead of FTOS is in the range of few microseconds. Of course, these numbers include in both approaches not the overhead required for network communication. The much better performance of FTOS in comparison to HTL can be explained by the virtual machine approach of HTL. While HTL uses a generic virtual machine that is also

[1] http://www.windriver.com/

used in Giotto, FTOS points out that it is practicable to generated an efficient run-time system.

The TMR example is used to point out the efficiency of the code. The application is implemented with a control rate of 400Hz. However, this control rate is admittedly at the upper bound. The clock synchronization is implemented on top of the network stack to avoid the use of specialized hardware. Therefore, the clock synchronization bound is in the range of some hundred microseconds. To achieve a significant lower bound, one must use specialized hardware. The limited clock resolution can be handled by using external hardware timers. Furthermore, it is useful to use a time-triggered communication protocol that implements the clock synchronization within the network stack. To implement these improvements, templates to support the Time-Triggered Protocol (TTP) [TTT03] are currently under development.

The elevator application points out the feasibility to generate code for distributed systems. Not all templates are currently implemented, but first prototypes are available. However, all the models are available so that the major benefits are apparent.

The generated source code consists of 12000 lines of code for one control unit of the elevator application. This number seems to be very high, but can be justified with the huge amount of driver code. The code for these drivers consists of more than 9000 lines of code. The footprint of the compiled run-time system is between 110 kB for the TMR application and 200 kB for the control unit of the elevator application.

The modeling complexity is kept to a minimum by the high abstraction level. The models for the TMR application consist of totally 105 objects (hardware model: 20, software model: 42, fault model: 26, fault-tolerance model: 17). For a better illustration of the complexity: the related XML files generated by the development tool consist of less than 200 lines of code. The code realizing the PID control function, which needs to be implemented by the developer using C as programming language, consists of only 21 lines.

In the context of the elevator application, the goal of a complete model-based development process is actually reached. The models consist of 336 objects (40, 215, 27, 54) and less than 700 lines of xml code. Besides these models that are used for the generation of the application's non-functional aspects, the application functionality is also modeled. For this purpose, another modeling tool called EasyLab[2] is used. The task functions are modeled by finite state automatons. Combining these tools results in an integrated model-based approach.

[2] http://easykit.informatik.tu-muenchen.de/

CHAPTER 6

Verification

Contents

Formal verification is an important topic for the development of fault-tolerant embedded systems [BFLS01]. Testing, as the standard validation technique, can only limit the failure probability to 10^{-4} failures per hour. The typical value for safety-critical systems that has to be achieved is however a mean time between failures (MTBF) of 10^9 hours [Sur94]. It is obviously impossible to gain confidence about such a system by testing [LS93].

Formal methods are mathematically based techniques that offer a rigorous way to analyze systems [CW96]. However, the application of these methods requires expertise in formal techniques. This is the major reason, why formal methods are not used to an extent, as it would be useful. The integration of formal methods into model-based development tools can minimize these drawbacks and help to improve the quality. The existing approaches such as the integration of formal methods in EsterelStudio [BKS03] or Matlab [BPT07] are usually applied at the modeling level. However this approach is for FTOS not applicable, since the model only defines the techniques, but does not dictate the concrete implementation. Therefore not only the models, but also the templates used for the code generation have to be verified. Since FTOS is based on a component/template-based approach, it is important to ensure that the components (generated from the templates) do not interfere in an unanticipated way and that the resulting behavior reflects the intended behavior [AK98a]. Unfortunately, the verification of a correct interaction between different components is very hard to achieve. Especially in fault-tolerant systems, the effects of possible faults must be carefully in-

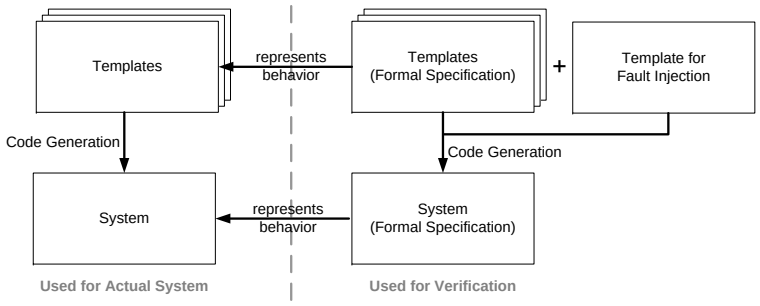


Figure 6.1: Approach for Formal Verification

vestigated, since the used components do not necessarily have to be based on the same fault assumptions.

The integration of formal verification in FTOS is by itself a topic for a separate thesis. Therefore, this chapter presents some general ideas on how to integrate formal verification. Grunske et al. [GKR05] identified two important issues that are the topic of this chapter:

- How to specify the failure behavior of a component, when its usage and environment are unknown?
- How to evaluate the safety properties for a system built with components?

This chapter presents answers to both questions. The fault hypothesis plays an important role in this process. Two components realizing the same functionality may be designed in different ways if they are based on different fault assumptions. Examples are the different solutions for consensus depending whether only fail-silent or arbitrary (Byzantine) faults must be tolerated. This chapter points out how template designer can formally specify the behavior of the developed templates. As these designers are usually no experts in formal methods, the goal in FTOS is to minimize the necessary expertise regarding formal verification. Based on the ideas of this section, a first prototype for a formal verification is integrated into FTOS. The prototype considers the behavior of generated components only and treats task functions as black box. The experiences with the prototype are presented at the end of this chapter.

6.1 Concept for Formal Verification

The main idea to integrate formal verification in FTOS is to define for each template, an additional template that specifies the formal behavior of the generated component in the presence of faults. One additional template is used to simulate the fault injection according to the fault model. Based on these formal specifications, the system and a formal specification of the system can be generated in parallel. The template developer can specify different formal assumptions and requirements that are evaluated by the use of a SMT[1] solver. The approach is depicted in Figure 6.1.

[1]Satisfiability Modulo Theories

For a first prototype, the interactions of the components are examined with respect to the behavior in the presence of faults. The application logic is not considered. The formal specification of the components can be based on the fault meta-model. It can be limited to the behavior concerning possible faults and the resulting errors (error propagation, error containment, error mitigation).

The major assumption for the validity of this approach is of course the consistency of the original template and the template describing the formal behavior. Therefore, the specification of the formal behavior of a component must be done thoroughly. To minimize the probability of inconsistency, the complexity of the specification has to be reduced. A major factor in the reduction of the complexity is the high level of abstraction in FTOS. Due to the specification of well-defined interaction interfaces of the run-time system's components and the time-triggered execution model, the possible error propagation is already restricted. Only the possible fault effects of hardware and software components that interact directly with the template/component have to be considered. If a developer designs for example a template for the scheduling component, the relevant fault effects are the fault states of the tasks, the ECU and its components. By using assertions, the developer can also specify several fault assumptions that were used during the implementation of a component . This helps to further minimize the modeling complexity.

Different approaches use for instance temporal logic properties to prove the correctness of fault-tolerant systems [BFS00, Lam02]. However, these approaches required good knowledge in formal methods. As the template developer will typically have only little knowledge in this area, this approach is not feasible. To reduce the required expertise, the main goal is to use a formal specification language that resembles common programming languages and does only introduce few additional concepts. BoogiePL [DL05] is such a specification language. It is an intermediate language for program analysis and program verification and resembles imperative programming languages. Very few concepts must be learned to understand the language and to be able to implement a formal specification. The additional constructs include an assertion command `assert` to specify conditions that must be satisfied by the system. The constructs `havoc` and `goto` allow the introduction of non-determinism. These constructs can be used to simulate the application behavior (e.g. the selection of a mode in the mode switch function), to introduce the activation of faults or to simulate uncertainty in the context of error propagation. By the command `assume`, verification paths can be aborted that are assumed to be not relevant. BoogiePL programs can be verified by using the SMT solver Z3 [dMB08].

6.2 Formal Specification of the System

This section discusses the translation of the different models into a formal model based on the formalism discussed in Chapter 2.

Hardware Model

The hardware model describes the set of hardware components $C_{HW} \in C$ used in the system. In the formal model, these components are treated as passive components. This means that each process of a hardware component Π_c with $c \in C_{HW}$ contains no actions. The variables V_c reflect the fault state of the hardware component and are explained later.

Software Model

The software model can be translated into a formal model to allow formal verification. The relevant set of components associated with the software model consists of components C_{Task} for each task and components that realize the run-time system. On each ECU e, there are three run-time components: the system component $c_{system,e}$, the communication component $c_{network,e}$, and the schedule component $c_{schedule,e}$. In the following, the actions and variables of these components are defined.

Each task component $c_t \in C_{Task}$ describes a task implemented by the application developer. As in the current version the task function is regarded as black box, internal variables and the concrete actions are not defined. It is sufficient to interpret the task component c_t as a component with **private ports** as interface variables $V_{c_t} = V_{t,read} \cup V_{t,write}$ and one action simulating the task function $a_t :: cpu_{assign,t.cpu} = t \rightarrow f_t(V_{t,read})(V_{t,write})$. For the current prototype, the function will simply relay erroneous input parameters. In addition, the task can also introduce new errors (erroneous results, deadline violation) according to the fault model. The guard specifies that the function can only complete its execution if the CPU is assigned to the task. Internal intermediate steps do not have to be considered due to the concept of simple tasks and the execution model.

The system component $c_{system,e}$ realizes the operational behavior as specified in Section 4.4.5. The component uses internal variables $V_{c_{system,e}} = V_{trigger,e} \cup V_{ports,e}$ to store trigger decisions and to realize the global ports. In addition, two variables are used to store the current micro time (v_δ) and the local time ($v_{time,e}$). The actions can be derived from the description of the operational semantics.

The network component $c_{network,e}$ realizes the communication between the different ECUs. The concrete implementation and actions depend on the communication protocol used. Its formal specification is omitted here. The provided interface variables $V_{c_{network,e},interface} = V_{rem\_priv\_ports,e}$ implement the remote private ports for each communication reference.

Finally, the scheduling component $c_{schedule,e}$ controls the assignment of available CPUs to tasks. The concrete behavior depends on the implementation, but several variables are defined to realize the interface. To set the status of the tasks concerning schedulability, the interface variable $v_{schedule,t} \in V_{schedule,Tasks}$ is used. Possible values are $Blocked$, $Started$ and $Completed$. The assignment of a CPU to a task is realized by the interface variables $cpu_{assign,t.cpu} \in V_{schedule,CPUs}$.

Fault Model

Based on the information contained in the fault model, the formal specification of the system can easily be augmented. This augmentation is carried out in three phases.

The first phase augments all existing components with additional interface variables $\mathtt{V}_{\mathtt{status}}$ describing the fault status of the component. This augmentation depends on the component type and the fault effects specified for this type. In addition, for each variable $\mathtt{v}_{\mathtt{p,C}}$ reflecting a private, global or communication port p within component C, a new variable $\mathtt{v}_{\mathtt{p_{status},C}}$ is added denoting the state of this port. Possible values for these port state variables are currently `Correct` and `Erroneous`.

In the second phase, the fault activation/injection is formally specified by an own component $\mathtt{C}_{\mathtt{FI}}$. The corresponding process $\Pi_{\mathtt{FI}}$ realizes the fault injection according to the fault assumptions. Internal variables $\mathtt{V}_{\mathtt{FI}}$ are used to realize constraints of the form $\mathtt{constraint} : \mathtt{V}_{\mathtt{FI}} \cup \mathtt{V}_{\mathtt{status}} \cup \{\mathtt{v}_{\mathtt{time}}\} \rightarrow \mathbb{B}$. The internal variables $\mathtt{V}_{\mathtt{FI}}$ contain information about the time of the last injected fault of a certain type. In combination with the current time contained in $\mathtt{v}_{\mathtt{time}}$, constraints regarding the period between the activation of different faults are realized. Also the number of currently active faults can be constrained by analyzing already existing faults in $\mathtt{V}_{\mathtt{FI}}$. The fault injection transitions are of the form: $\mathtt{transition} : \mathtt{V}_{\mathtt{FI}} \cup \{\mathtt{v}_{\mathtt{time}}\} \rightarrow \mathtt{V}_{\mathtt{FI}} \cup \mathtt{V}_{\mathtt{status}}$. Faults can be activated or transient faults can be removed from the system. The actions contained in $\Pi_{\mathtt{FI}}$ are then of the form $\mathtt{constraint} \rightarrow \mathtt{transition}$ and comply with the assumptions of the fault model.

The last step is the definition of the error propagation. For each component, the existing actions of the different components are augmented with respect to the guard and/or the transition to reflect the behavior in the context of faults/errors. In addition, also new actions are introduced. The definition of the error propagation for a component depends on the own fault state and of the fault states of the interacting components and relevant ports that are read. The augmentation is of course implementation dependent and has to be defined by the template developer.

Fault-Tolerance Model

The addition of fault-tolerance mechanisms expands the system and the communication components. A set of variables is added to the system components. It contains Boolean variables reflecting the test results. Additional variables are created for the realization of checkpoints and to reflect the master decision of replica management objects. All these variables are only used for the realization of the fault-tolerance mechanisms and do not affect the standard operation of the error-free system.

In addition, the system components are augmented with additional actions realizing the test execution, the error treatment, the integration and the pro-active operations as discussed in Section 4.6.4. Test actions update the Boolean test variables. A change of the fault configuration, denoted in the status variables, can trigger the execution of an error reaction. An error reaction is, similar to standard actions, an action of the form $\mathtt{r} : \langle \mathtt{grd} \rangle \rightarrow \langle \mathtt{stmt} \rangle$. The guard $\mathtt{grd} = \mathtt{trigger} \wedge \mathtt{grd}'$ of this action is a combination of a `trigger` statement specifying the enabling change of the fault configuration and a

guard grd′ specifying an enabling condition based on the application logic. Thus, the reaction depends on the previous and the active fault configuration, as well as on the current application status and in particular the program counter. The statement stmt updates one or more variables of the system. The concrete actions can be derived from the operational semantics.

In addition to the system component, also the network component is augmented to support integration and liveness tests. An additional set of variables is used to signal integration requests, the successful completion of the integration and to store the receive time of the last message from observed control units. The concrete actions depend on the implementation of the component. The schedule component is left unchanged.

6.3 Prototype

This section discusses a first prototype. For reasons of simplicity, this prototype ignores the different fault effects and only distinguished between correct and erroneous components. Additional assumptions such as a simplified handling of time are used to quickly develop the first prototype. The main idea of the prototype is to demonstrate the general feasibility of this approach, but also to verify the validity of the consensus protocol used in the context of the triple-modular redundancy application.

General Framework

The formal verification is based on a generic framework that reflects the error state of the different hardware and software components and the time. For each hardware or software component c, a variable s_c denotes the error state. In general, the values for this status variable depend on the component type. The basic set of possible values is defined in the fault meta-model, but further values for different types are defined to reflect error propagation. This can be illustrated using ports as example. In the model, ports represent a logical component that cannot be affected by a fault directly. However, a port can of course be affected by error propagation through corrupted memory or by a lost network message. Therefore, additional error states have to be defined for each component type. In the current version, only Boolean error states are used.

In addition to the error status of the individual components, the generic framework represents a variable τ that reflects the time. The local time for each ECU is reflected by internal variables of the concrete clock-related components. The definition of the fault injection function is more complex. Figure 6.2 shows the begin of the template defining the fault injection function. The function is executed in three phases. Within the first phase, the status of each variable is randomly set to a new fault state using the havoc command under consideration of the LTBF and failure mode constraints. In the second phase, the state of each FCU is calculated. The active fault configuration is determined in phase 3. Paths with fault configurations that are not considered in the fault model are aborted by the use of the assume command. In general, the activation of faults is of course a non-deterministic process. To achieve a realistic mapping, the fault injection

```
implementation faultInjection(sec: int, nsec: int)
{
«FOREACH faultModel.fcu.eAllContents.typeSelect(ComponentBehavior) AS cb-»
«IF cb.expectedFailures.first().mode==FailureMode::TRANSIENT-»
«FOREACH cb.components AS c»
    goto FAULT_«c.name», CORRECT_«c.name», KEEP_«c.name»;
FAULT_«c.name»: //insert fault
    assume «getStatusVariable(c)»;
    «getStatusVariable(c)»:=false;
    «getInjectionTimeVariableSeconds(c)»:=sec;
    «getInjectionTimeVariableNanoseconds(c)»:=nsec;
    goto END_«c.name»;
CORRECT_«c.name»: //remove fault
    assume !«getStatusVariable(c)» &&
        (((sec-«getInjectionTimeVariableSeconds(c)»)*1000000000)+nsec-
            «getInjectionTimeVariableNanoseconds(c)»>=«cb.expectedFailures
    «getStatusVariable(c)»:=true;
    «getInjectionTimeVariableSeconds(c)»:=sec;
    «getInjectionTimeVariableNanoseconds(c)»:=nsec;
    goto END_«c.name»;
```

Figure 6.2: Formal Specification of Fault Injection Function

can be implemented as a parallel process. However, this approach would lead to a dramatic state explosion inside the verification tool. Therefore, the approach is based on a manual fault injection. The template designer can call the fault injection function, whenever a change of the error status of a component might have effects.

Component Specification

The formal specification of the different templates for the run-time system components is straightforward. Based on the variables denoting the error state, the developer can specify the behavior of the component and the effect on the error state of involved components. By using the `assert` statement, the developer can state conditions that have to be satisfied by the system. In the context of the voting algorithm, the relevant component tests whether a consistent decision is reached. The effects of the component execution on the local and global time can be specified by using the relevant variables. This is important to make use of the Least Time Between Failure assumptions stated in the fault model.

Specification of Application-Specific Requirements

It should also be possible to evaluate application specific requirements besides the requirements that are listed in the formal specification of the run-time system components. A typical requirement is that no output is performed based on an erroneous port value. However, these requirements cannot be specified application-independent, since some applications might also tolerate a temporarily erroneous output operation. To specify the application specific requirements, the application developer currently has to add the assumptions at the correct position within the generated formal specification. A more developer-friendly possibility to state such requirements within the modeling phase is future work.

Experiences

Using the formal verification introduces an easy mean to verify the correct selection of appropriate fault-tolerance mechanisms in the context of a specific fault hypothesis. For most of the implemented components, one can find a solution in the literature with a correctness proof. However, these proofs are usually based on a specific fault hypothesis and it remains unclear whether some modifications might jeopardize the correctness of the solution. In the concrete example of the voter, it was not clear how many communication faults could be accepted by the presented solution for voting and how other faults would influence the correctness. The prototype allows an easy and efficient way to get the correct numbers.

Based on the formal specification of the templates, a formal model of the whole application is generated. This model contains about 1000 lines of code and consists of about 150 different variables for the TMR application. The high abstraction level and the determinism of the application in the absence of faults lead to an optimal performance of the verification process and avoid the common problem of state explosion. Using the very efficient SMT solver Z3, the correctness of the system could be proved in negligible time (less than one second). These numbers demonstrate that the approach is feasible. Of course, the required time for the verification process will increase if sophisticated error states instead of Boolean error states are used. One drawback of BoogiePL and Z3 is the concept to perform the verification at procedural level. While this helps to keep the verification process small and fast, the counter examples are only presented for one component. Nevertheless, to be really useful, the counter example should include information about the whole trace through the involved components. Therefore, the prototype currently combines the formal behavior of all components into one block and verifies the complete block. The only exception is the fault injection function. This function is treated separate, because inlining would lead to a dramatic increase of the generate lines of code. The applied abstractions of Z3 are another problem. Loops for example are analyzed in a pessimistic way. To reach a useful result, one must unroll the loop if a requirement should be verified that requires the verification of the system executing several loop cycles.

In summary, the experience with the first prototype is positive. Due to the clear concepts of FTOS and the ease of learning BoogiePL, the integration of formal methods is straightforward. Besides the formal verification of different properties, a major benefit of the formal specification is the increased understanding in the complete system. This helps not only to design correct code, but to also implement the system more efficiently.

Besides the integration of sophisticated error types, the next prototype will focus on the presentation of counter examples. These counter examples are currently presented at the level of BoogiePL. However, to be really useful, the feedback should be performed at the modeling level.

CHAPTER 7

Conclusion

This thesis discusses the design of a model-based development process including automatic code generation for fault-tolerant real-time systems. The major focus of the developed tool FTOS lies on the generation of code realizing non-functional aspects. These aspects, such as fault-tolerance mechanisms, communication within the distributed system, and scheduling, cause most of the software complexity of these systems. The presented approach complements existing tools in an optimal way, in the sense that existing tools focus predominantly on functional aspects. The predicted benefits of the approach are pointed out using two demonstrators. A system controlling an elevator that consists of a hot-standby system to realize the application logic and several field controllers to perform I/O operations is implemented without writing a single line of code by using FTOS and another tool used to model the application functionality. Another demonstrator proves the feasibility of the approach, if hard real-time requirements with control response time in the range of few milliseconds have to be achieved.

The efficiency of the code and the possibility to generate code for heterogeneous platforms is obtained by using a template-based code generator. Templates realize individual solutions for a certain aspect of the system and a set of platforms. The code generator selects the relevant templates, adapts these templates according to application requirements, and can therefore tailor an optimized run-time system.

The major contribution of this thesis is the formulation of an adequate modeling language. This language is on the one hand abstract enough to simplify the modeling process, but has on the other hand explicit execution semantics to support the code generation. The unique feature of FTOS is the addition of fault-tolerance in a complete model-based way. Other approaches based on components require that the application developer selects and configures the concrete components. This typically requires good knowledge about implementation details. In contrast, FTOS is designed both for experts and novices in the field of fault-tolerant systems. Experts benefit from the ex-

tensive code generation; novices from the possibility to model fault-tolerant systems at a very high level. Furthermore, the different parameters for the fault-tolerance mechanisms help to avoid typical pitfalls during the design of such systems.

To achieve a system-oriented approach, the modeling language combines several different aspects. It allows the definition of the hardware architecture, the software components, their interaction and timing constraints, the fault assumptions, and the selection of fault-tolerance mechanisms.

The software model is based on the concept of logical execution time. For each execution of a component, the user specifies the logical start and stop times. This paradigm helps to abstract from low-level details on the implementation of the scheduling, but guarantees a deterministic execution of the system. The thesis points out that the implementation of the various mechanisms required for fault-tolerant systems in a generic way is simplified by exploiting this paradigm. Amongst others, the thesis discusses implementations of the major fault-tolerance mechanisms such as passive, semi-active and active replication, rollback, and rollforward recovery. In addition, a scheduling algorithm was introduced that combines the optimality of Earliest-Deadline-First with the robustness of Cyclic Executive.

To cope with the issue of different fault assumptions, the modeling language forces the user to formally specify the assumptions using a generic description. This approach allows the appropriate selection of adequate templates and allows the integration of formal methods to prove the overall correctness. A first prototype is implemented, that integrates formal verification based on a SMT (Satisfiability Modulo Theories) solver in FTOS.

Future Work

An obvious point for future work is to continue with the integration of formal methods. The current prototype is based on very simple fault effects and was primarily designed to prove the overall concept. Future work will be on the integration of both sophisticated fault effects and on the integration of the application functionality, which is currently regarded as black box.

Another important future work is the use of FTOS in industrial applications. Currently, one application in the context of railway systems is implemented with FTOS. The used fault-tolerance mechanisms are based on a triple-modular-redundancy architecture. Initial results indicate a significant reduction of development time.

A very important issue in computer science in general is multicore computing. The advantages of multicore computing in the area of dependability are obvious. The application tasks can be separated physically and executed on distinct cores. Amongst others, this can be exploited to execute software with different criticality levels on one processor. However, the research in this area is not mature and a lot of interesting and challenging questions have to be answered. Nevertheless, the tool FTOS is already designed to support multicore computing, so that it should be easily possible to support multicore processors. The design of adequate concepts and templates to exploit the new features of such architectures is a very interesting topic for future research.

APPENDIX A

Related Work - Modeling Tools, Code Generators and Frameworks

Several modeling languages and code generators target the area of embedded systems. The tools differ in the number of covered aspect, such as application logic, hardware architecture, deployment diagrams and timing aspects, the tool support and the code generation possibilities. In addition, there are some frameworks that are targeted for fault-tolerant systems. The following sections give an overview of existing tools and reference the related work of this thesis.

A.1 Unified Modeling Language

Certainly the most important modeling language is the **Unified Modeling Language (UML)** [Obj07b]. UML was initially designed for standard information systems, but was recently adapted for modeling embedded, fault-tolerant real-time systems by the introduction of adequate profiles.

The UML Profile for **Modeling Quality of Service and Fault-Tolerance Characteristics and Mechanisms** [Obj04a] provides several frameworks to describe fault-tolerant systems, quality of service and risk assessment. The focus of the fault-tolerance framework is to model software redundancy/replication. The risk assessment framework allows the identification of potential threats and weaknesses, unwanted incidents, risk mitigation and treatments.

Timing and scheduling aspects can be described by using the **Schedulability, Performance and Time (SPT)** profile [Obj04b]. The profile covers the modeling of resources, time, concurrency, schedulability and performance. Within the European project High Integrity Distributed Object-Oriented Real-Time Systems (HIDOORS) [MLJ03, VSWH02], SPT profile compliant mechanisms for modeling safety-critical, em-

bedded real-time applications were introduced. The new profile MARTE [Obj07c] is intended to replace the SPT profile.

Several other projects such as OMEGA [Hoo02], FLEXICON [MEG+04], or books [Dou99, IC02, LMS03, Mar06] presented solutions for modeling embedded real-time systems using UML.

However, all of these solutions focus on specific aspects and do not cover all necessary aspects for modeling fault-tolerant real-time systems [ZBL07]. Furthermore, there are no adequate code generators available to support code synthesis for more sophisticated models than class diagrams and state charts [KGG+06]. The reason is the lack of precise semantics of the UML models [JSEB04, BGP07].

A.2 Domain-Specific Tools

A.2.1 Commercial Tools

A number of tools are available that promote the model-based design of embedded software including code generation. However, these tools focus mainly on the functional aspects.

Matlab/Simulink [Bar05] by Mathworks focuses on the development of algorithms for data visualization, analysis and numeric computation. C code can be generated by using the Real-Time Workshop. Features of distributed systems like networking or process management are not covered by the Real-Time Workshop.

SCADE [DSMG04] by Esterel Technologies is a tool for code generation of safety-critical applications based on synchronous languages like Esterel[BG92] or Lustre [CPHP87]. The main idea of SCADE is to generate sequential code to avoid problems like race conditions. Therefore, features like networking and process management are not covered.

Statemate [HLN+90] provides a set of modeling languages to specify the application functionality. The most important concepts are statecharts and data flow graphs. Non-functional aspects or the realization of a distributed execution are not supported by the code generator.

Due to the focus on functional aspects, the tools are the prefect counterpart for the developed tool FTOS. FTOS can be used to design the middleware to execute the application code developed with the mentioned tools. One remaining problem is the request for referential transparency for all task functions. Here, it is necessary to avoid modeling constructs that lead to internal variables in the generated code. The integration of FTOS and SCADE is currently in progress.

A.2.2 Tools from Academia

The design of the FTOS model was significantly influenced by the **Ptolemy** project [Lee03] from UC Berkeley. Ptolemy describes different models of computation, including their combination (modal models) and code generation support. FTOS adopted

the concept of actor-oriented design from Ptolemy and examined different models of computation.

Giotto[HHK03], another project from UC Berkeley, is used for the platform-independent specification of real-time systems. It is based on the concept of Logical Execution Times and the simple task model [Kop97]. Tasks are used to realize the application logic and are executed periodically. The communication between tasks is realized by local ports (ports attached to a task) and drivers. Drivers realize the communication between different ports and are executed in logically zero time. The code for drivers has to be implemented by the developer. Similarly, the developer can also implement sensor and actuator drivers that realize the communication with the environment. To allow a flexible design, Giotto introduces the concept of modes and guards. In contrast to FTOS, only one mode can be active at a time.

Virtual machines are used to reach the goal of platform independence. The result of the compilation process of the Giotto program is embedded code (E-Code) [HKMM02]. This E-Code is executed on a corresponding virtual machine, the E-Machine. For scheduling, the E-Machine may pass the tasks to the scheduler of the operating system. Alternatively, it may use another virtual machine, the S-Machine, for scheduling. The S-Machine interprets scheduling code (S-Code) [HKM03]. The approach has the advantage that the platform independent application execution (E-Code) and physical execution (S-Code) are separated [KSH05].

The **Timed Definition Language (TDL)** [FFPT05, FP07] is a successor of Giotto and adds a component model and full support for distribution. The component model allows the decomposition of the whole program into distinct subprograms. The different components, called modules, can have different active modes, thus resolving the restriction of Giotto. However, a module is limited to one hardware node. Mode changes affecting several nodes have to be realized by the developer. To simplify the application design, the code for drivers is generated in TDL by the design tools. However, sensor and actuator functions must still be implemented by the application developer.

To support distribution, TDL allows a mapping of modules to hardware nodes, denoted as electronic control units (ECU) and generates code for the time-triggered communication protocol FlexRay [Far06].

The **Hierarchical Timing Language (HTL)** [GHI+06] is another successor of Giotto. It adds similar to TDL the concepts of modules to support different modes to be active at a time. However, these modules are again restricted on one node. Communicators are used to realize the communication between the different modules. This concept resembles the concept of global ports used in FTOS. The communication between different computational nodes should be supported by the tool, but must be currently implemented by the developer.

A.3 Meta-Modeling Frameworks

Recently, meta-modeling frameworks, such as **openArchitectureWare (oAW)** [VSK05], the **General Modeling Environment (GME)** [LMB+01], or **MetaEdit** [Tol04b, met07],

became very popular for developing embedded software [Tol04a, SV06]. Meta-Modeling Frameworks allow the specification of domain-specific meta-models that can be used to specify concrete models describing a concrete application. Typical examples for the used meta-modeling language are OMG's MetaObject Facility (MOF) [Obj06] used by GME or the Eclipse Modeling Framework (EMF) [BSM+03] used by oAW. The meta-modeling languages are usually based on the UML class diagram notation. It is possible to constrain the design possibilities by defining rules. For this purpose, languages to define constraints, such as OMG's Object Constraint Language (OCL) [Obj03] or the CHECK language [EFH+07] in oAW, are incorporated into the frameworks.

Although the projects in the context of meta-modeling frameworks demonstrate the advantages of the approach, there are only view projects targeting fault-tolerant systems. **AIM (Architecture Information Modeling)** [RP07] is a generic, extensible platform for modeling software and systems in multi-team environment with a particular focus on the high integrity real-time systems domain. AIM specifies its own meta-modeling framework based on MOF. The developers can specify different meta-models, e.g. a meta-model describing the system architecture or a meta-model describing the implementation of the system in ADA. The meta-models can be reused in different projects. The main focus of the project is on the construction of a consistent database for the system models, the integration of different views [iee00, MEH01] including their graphical notation and the definition of rules to maintain and check consistency of different views. Since no concrete meta-models are pre-defined, the integration of code generators is not possible.

A.4 Code Generators

Whalen and Heimdahl [WH99b, WH99a] identified the requirements for high-integrity code generation. However, these requirements are predominantly of academic nature. For example, the requirement that both the model and the code must have well-defined syntax and semantics raises a major challenge as there are hardly any production-ready programming languages with well-defined semantics. Even the authors admit that this is a major problem. Their solution is to use an automaton based specification language [HL96], similar to statecharts [Har87]. As target language, a very small subset of standard imperative programming languages called SIMPL is used. Only the most essential operations like constructs for variables, constants, functions, procedures, basic and composite (array and record) types. Control flow statements such as break, goto, continue, or early return and pointers are not allowed [WH99b]. Thus, the potential of code generation using this target language is very restricted. Especially system level aspects can not be implemented using these languages.

A.5 Fault-Tolerance Infrastructures

Different research projects proposed frameworks for the development of fault-tolerant systems. **Chameleon** is a software implemented fault tolerance (SIFT) infrastructure

to achieve reliability in a heterogeneous, distributed system [KBWI99]. The intended systems are networked applications. The major goals are the provision of run-time adaptability and the support of different criticality requirements. The fault-tolerance is achieved by ARMOR (adaptive reconfigurable mobile objects for reliability) components that can be added to a non-fault-tolerant application. Different fault-tolerant configurations can be achieved by supporting a fine-grained composability. Techniques such as control flow signatures [MM88], data signatures or smart heartbeats are used for error detection. A four-level detection hierarchy is used to optimize the error detection and the related overhead. The levels range from inter-component error detection (Level 1) to error detection between replicated ARMORs distributed on different nodes (Level 4). Different optimization techniques can be applied to get a trade-off between overhead and error detection latency.

While some goals are shared with this thesis, there are however different drawbacks. The software components realizing the application functionality have to be adapted to be used in combination with the ARMORs. This leads to a mix of the application functionality and the fault-tolerance aspects. The achieved error detection latency in the range of several seconds is not acceptable in the context of real-time systems and there are no mechanisms to guarantee a timely predictable behavior.

The **Delta-4** approach [PCD91] emphasizes the tight relationship between distribution and fault-tolerance. The approach is restricted to the replication of software components that share no common memory and interact using explicit messages. The basic units of fault-tolerance are nodes of a distributed computing system. Besides fail-silent faults, also fail-uncontrolled faults are considered. Fail-silence with respect to faults in the time domain is achieved by special hardware components, called network attachment controllers (NAC), similar to TTP. Faults in the value domain can be tolerated by implementing voters and using active replication (hot-standby). In case of fail-silent nodes, Delta-4 proposes the use of either active, passive (cold-standby) or semi-active replication. The disadvantages of Delta-4 are the necessity to use special hardware components and the restriction on hardware replication to achieve fault-tolerance.

The **Generic Upgradable Architecture for Real-time Dependable** (**GUARDS**) [PABD+99] presents a component-based approach to achieve fault-tolerance. The project proposes an architecture that is suited for dependable systems. The implementation of the mechanisms has to be realized by the application developer. The process is supported by a development and validation environment. It is based on software implemented fault-tolerance to cope with the unreliability of underlying COTS components. The fault-tolerance mechanisms are based on a two-level replication: replicated hosts using shared memory for communication and second level using an interchannel communication network (ICN). The fault-tolerance mechanisms are constrained on this replication mechanism, which restricts the flexibility of this approach. Model-based techniques are used for the evaluation of the system. The design tool HRT-HOOD addresses the problem of executing replicated components within a distributed system [BW94]. For GUARDS, HRT-HOOD was expanded by the concept of virtual nodes to support the design of distributed systems. The generation of code is partially supported, e.g. by the generation of exchange tables. However, the framework does not integrate the techniques into one common approach, but uses

them separately. The only linking between the dependability and the real-time models are written documents [BFLS01].

The project **Distributed Embedded ARchitecture using COTS components DEAR-COTS** [VCP+00, PVW04] proposes the use of a component based architecture. Using a simple and transparent programming model, the low-level implementation details of distribution and replication are abstracted from the programmer. The replication of software components is supported by the use of the timely computing base (TCB) [VC02] and timed messages [PBWB00]. Using TCB, a deterministic execution of the replicated components can be guaranteed, even in preemptive multitasking environments. The details required for replication and distribution must be specified in a second configuration phase. **Components** are used to structure the replication units and to allow system configuration. Components can be distributed over several hosts and be replicated. Shared data objects handle the different versions of data objects based on the release time of the data to guarantee replica determinism. A replica manager handles the monitoring of the different replicated objects and is also responsible for consistency. The techniques to reach a consistent result out of redundant results are not specified in DEAR-COTS and left to the implementation of the replica manager. The communication manager provides appropriate communication algorithms [PV00]. The selection and configuration of these components must be done by the application developer in a second phase. An automated selection using a model-based approach is not intended.

Another weakness is the restriction on the problem of fault-tolerance aspects. Heterogeneous platforms are not supported by DEAR-COTS [PVW04]. Like the other approaches, DEAR-COTS is restricted to replication to achieve fault-tolerance and does not support the toleration of application design faults.

The **Time-Triggered Architecture (TTA)** [KB03] focuses on fault-tolerant communication. TTA is a framework for the design and implementation of distributed fault-tolerant applications with a focus on the automotive and aviation industry. TTA provides different services such as predictable communication with small latency, clock synchronization and membership service [KGR91]. The approach is based on a hardware solution, the so-called TTP/C controller [TTT03], running the TTP protocol. TTP realizes time-triggered communication on redundant communication channels. Because TTA concentrates only on fault-tolerant communication, the implementation of mechanisms for the toleration of other error sources has to be done by the developer itself. Another disadvantage of TTA is the restriction on specialized hardware.

Bibliography

[ABPG05] Colin Atkinson, Christian Bunse, Christian Peper, and Hans-Gerhard Gross. *Component-Based Software Development for Embedded Systems - An Introduction*. Number 3778 in Lecture Notes in Computer Science. Springer, 2005.

[AC77] Algirdas Avizienis and Liming Chen. On the implementation of n-version programming for software fault-tolerance during program execution. In *Proceedings of the Annual IEEE International Computer Software and Applications Conference*, Nov 1977.

[AG93] Anish Arora and Mohamed Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027, Nov 1993.

[Agh86] Gul A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, 1986.

[AK98a] Anish Arora and Sandeep S. Kulkarni. Designing masking fault-tolerance via nonmasking fault-tolerance. *IEEE Transactions on Software Engineering*, 24(6):435–450, Jun 1998.

[AK98b] Anish Arora and Sandeep S. Kulkarni. Detectors and correctors: A theory of fault-tolerance components. In *International Conference on Distributed Computing Systems*, pages 436–443, 1998.

[ALR01] Algirdas Avizienis, Jean-Claude Laprie, and Brian Randell. Fundamental concepts of dependability. Technical report, LAAS-CNRS, April 2001.

[ALRL04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[AMST97] Gul A. Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, Jan 1997.

[AS85] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, 1985.

[Atm07] Atmel, San Jose, CA. *AT90CAN32/64/128 Datasheet*, Revision F edition, Nov 2007.

[Avi82] Algirdas Avizienis. The four-universe information system model for the study of fault-tolerance. In *International Symposium on Fault-Tolerant Computing 12*, pages 6–13, Santa Monica, CA, Jun 1982.

[Bak91] Theodore P. Baker. Stack-based scheduling for realtime processes. *Real-Time Systems*, 3(1):67–99, 1991.

[Bar05] Paul Barnard. Software development principles applied to graphical model development. In *AIAA Modeling and Simulation Technologies Conference and Exhibit, San Francisco*, August 2005.

[BCE+03] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, January 2003.

[BCGG97] Andrea Bondavalli, Silvano Chiaradonna, Felicita Di Giandomenico, and Fabrizio Grandoni. Inter-channel state restoration. Activity Output, ESPRIT Project 20716 GUARDS Report I1-SA4/6006/B, GUARDS Project - PDCC, November 14 1997.

[BFLS01] Andrea Bondavalli, Alessandro Fantechi, Diego Latella, and Luca Simoncini. Design validation of embedded dependable systems. *IEEE Micro*, 21(5):52–62, 2001.

[BFS00] Cinzia Bernardeschi, Alessandro Fantechi, and Luca Simoncini. Formally verifying fault tolerant system designs. *The Computer Journal*, 43(3):191–205, 2000.

[BFVY96] Frank J. Budinsky, Marilyn A. Finnie, John M. Vlissides, and Patsy S. Yu. Automatic code generation from design patterns. *IBM Systems Journal*, 35(2):151–171, 1996.

[BG92] Gérard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[BGG+98] Andrea Bondavalli, Felicita Di Giandomenico, Fabrizio Grandoni, David Powell, and Christophe Rabejac. State restoration in a cots-based n-modular architecture. In *Proceedings of the 1st IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 174–183, Kyoto, Japan, April 1998. IEEE Computer Society.

[BGP07] Christian Bunse, Hans-Gerhard Gross, and Christian Peper. Applying a model-based approach for embedded system development. In *EUROMICRO '07: Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO 2007)*, pages 121–128, Washington, DC, USA, 2007. IEEE Computer Society.

[BHLM94] Joseph Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt.

Ptolemy: A framework for simulating and prototyping heterogenous systems. *International Journal in Computer Simulation*, 4(2):155–182, 1994.

[BHMR95] Roberto Baldoni, Jean-Michel Helary, Achour Mostefaoui, and Michel Raynal. On modeling consistent checkpoints and the domino effect in distributed systems. In *Proceedings of the IEEE International Conference on Future Trends in Distributed Computing Systems*, pages 314–323, 1995.

[BKS03] Gérard Berry, Michael Kishinevsky, and Satnam Singh. System level design and verification using a synchronous language. In *ICCAD '03: Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design*, pages 433–440, San Jose, CA, USA, 2003. IEEE Computer Society.

[BKS05a] Christian Buckl, Alois Knoll, and Gerhard Schrott. Development of Dependable Real-Time Systems with Zerberus. In *11th International Symposium, Pacific Rim Dependable Computing, PRDC 2005*, pages 404–408, Changsha, China, Dec 2005. IEEE.

[BKS05b] Christian Buckl, Alois Knoll, and Gerhard Schrott. The zerberus language: Describing the functional model of dependable real-time systems. In *Dependable Computing, Second Latin-American Symposium, LADC 2005*, Lecture Notes in Computer Science, pages 101–120, Salvador, Brazil, Oct 2005. Springer.

[BKS06a] Christian Buckl, Alois Knoll, and Gerhard Schrott. Model-based development of fault-tolerant embedded software. In *Second International Symposium on Leveraging Applications of Fomal Methods, Verification and Validation (IEEE-ISoLA)*, pages 113–120, Paphos, Cyprus, 2006.

[BKS06b] Christian Buckl, Alois Knoll, and Gerhard Schrott. Template-based development of fault-tolerant embedded systems. In *International Conference on Software Engineering Advances, ICSEA 2006*, Tahiti, French Polynesia, Oct 2006. IEEE.

[BPT07] Andreas Bauer, Markus Pister, and Michael Tautschnig. Tool-support for the analysis of hybrid systems and models. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 924–929, San Jose, CA, USA, 2007. EDA Consortium.

[BS88] Theodore P. Baker and Alan C. Shaw. The cyclic executive model and ada. In *IEEE Real-Time Systems Symposium*, pages 120–129, Huntsville, Alabama, USA, Dec 1988.

[BS05] Bruno Bouyssounouse and Joseph Sifakis, editors. *Embedded Systems Design - The ARTIST Roadmap for Research and Development*, volume 3436 of *Lecture Notes in Computer Science*. Springer, 2005.

[BSM+03] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy J. Grose. *Eclipse Modeling Framework*. Prentice Hall International, Oct 2003.

[BSW+00] Saurabh Bagchi, Balaji Srinivasan, Keith Whisnant, Zbigniew Kalbarczyk, and Rafishankar K. Iyer. Hierarchical error detection in a software implemented fault tolerance (sift) environment. *IEEE Transactions on Knowledge*

and Data Engineering, 12(2):203–224, Mar 2000.

[But05] Giorgio C. Buttazzo. Rate monotonic vs. edf: judgment day. *Real-Time Systems*, 29:5–26, 2005.

[BW94] Alan Burns and Andy J. Wellings. HRT-HOOD: a structured design method for hard real-time systems. *Real-Time Systems*, 6(1):73–114, Jan 1994.

[CC89] Houssine Chetto and Maryline Chetto. Some results of the earliest deadline scheduling algorithm. *IEEE Transactions on Software Engineering*, 15(10):1261–1269, Oct 1989.

[CF94] Flaviu Cristian and Christof Fetzer. Probabilistic internal clock synchronization. In *Symposium on Reliable Distributed Systems*, pages 22–31, 1994.

[COR94] CORPORATE IEEE, Institute of Electrical and Electronics Engineers, Inc. Staff. *IEEE Standard for Information Technology - Portable Operating System Interface (POSIX): System Application Program Interface (API), Amendment 1: Realtime Extension (C Language), IEEE Std 1003.1b-1993*. IEEE Standards Office, New York, NY, USA, 1994.

[CPHP87] Paul Caspi, Daniel Pilaud, Nicholas Halbwachs, and John A. Plaice. Lustre: a declarative language for real-time programming. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 178–188. ACM Press, 1987.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[CW96] Edmund M. Clarke and Jeannette M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[DHSZ03] Kevin Driscoll, Brendan Hall, Hakan Sivencrona, and Phil Zumsteg. Byzantine fault tolerance, from theory to reality. In *Computere Safety, Reliability and Security: 22nd International Conference, SAFECOMP 2003*, volume 2788 of *Lecture Notes in Computer Science*, pages 235–248. Springer Verlag, Edinburg, UK, September 2003 Proceedings 2003.

[DL05] Robert DeLine and K. Rustan M. Leino. Boogiepl: A typed procedural language for checking object-oriented programs. Technical report, Microsoft Research, Mar 2005.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Budapest, Hungary, April 2008.

[Dou99] Bruce Powel Douglass. *Doing Hard Time - Developing Real-Time Systems with UML, Objects, Frameworks and Patterns*. Addison-Wesley, 1999.

[DSMG04] Bernard Dion, Thierry Le Sergent, Bruno Martin, and Herbert Griebel. Model-based development for time-triggered architectures. In *Digital Avionics Systems Conference, 2004. DASC 04. The 23rd*, volume 2, pages 6.D.3– 6.1–7, 2004.

[EFH+07] Sven Efftinge, Peter Friese, Arno Haase, Clemens Kadura, Bernd Kolb, Di-

eter Moroff, Karsten Thoms, and Markus Voelter. *openArchitectureWAre User Guide*, Sep 2007.

[EVHK] Sven Efftinge, Markus Volter, Arno Haase, and Bernd Kolb. *openArchitectureWare*.

[Far06] Emilia Farcas. *Scheduling Multi-Mode Real-Time Distributed Components*. PhD thesis, Department of Computer Sciences, University of Salzburg, Jun 2006.

[FFPT05] Emilia Farcas, Claudiu Farcas, Wolfgang Pree, and Josef Templ. Transparent distribution of real-time components based on logical execution time. In Yunheung Paek and Rajiv Gupta, editors, *ACM SIGPLAN/SIGBED 2005 Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 31–39. ACM, 2005.

[FGHL04] Peter H. Feiler, David P. Gluch, John J. Hudak, and Bruce A. Lewis. Embedded systems architecture analysis using sae aadl. Technical report, Carnegie Mellon University, 2004.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[FM98] Emmerich Fuchs and Dietmar Millinger. Task set design tools for an embedded distributed control system. In *EW 8: Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*, pages 182–188, New York, NY, USA, 1998. ACM Press.

[Foh95] Gerhard Fohler. Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. In *RTSS '95: Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS '95)*, pages 152–161. IEEE Computer Society, 1995.

[FP07] Claudiu Farcas and Wolfgang Pree. A deterministic infrastructure for real-time distributed systems. In *OSPERT 2007 Workshop on Operating Systems Platforms for Embedded Real-Time applications*, 2007.

[GHI+06] Arkadeb Ghosal, Thomas A. Henzinger, Daneil Iercan, Christoph M. Kirsch, and Alberto Sangiovanni-Vincentelli. A hierarchical coordination language for interacting real-time tasks. In *Proceedings of the 6th ACM International Conference on Embedded software*, pages 132–141, Seoul, Korea, Oct 2006. ACM.

[GKR05] Lars Grunske, Bernhard Kaiser, and Ralf H. Reussner. Specification and evaluation of safety properties in a component-based software engineering process. In Collin Atkinson, editor, *Component-Baseed Software Development*, volume 3778 of *Lecture Notes in Computer Science*, pages 249–274. Springer Verlag, 2005.

[GR83] A. Goldberg and D. Robson. *SmallTalk 80: The Language and its Implementation*. Addison-Wesley, 1983.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[Hen04] Thomas A. Henzinger. Embedded software: Better models, better code. In *Applications and Theory of Petri Nets 2004*, number 3099 in Lecture Notes in Computer Science, pages 35–36, Jun 2004.

[HG05] David W. Humphrey and Asish Ghosh. Siemens' process safety systems deliver modern features on a proven platform. Technical report, ARC Advisory Group, Dedham, MA 02026 USA, Oct 2005.

[HHK03] Thomas Henzinger, Ben Horowitz, and Christoph Kirsch. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91:84–99, January 2003.

[HKM03] Thomas A. Henzinger, Christoph M. Kirsch, and Slobodan Matic. Schedule-carrying code. In *Proceedings of the Third International Conference on Embedded Software (EMSOFT)*. Lecture Notes in Computer Science, Springer-Verlag, 2003.

[HKMM02] Thomas Henzinger, Christoph Kirsch, Rupak Majumdar, and Slobodan Matic. Time safety checking for embedded programs. In *Second International Workshop on Embedded Software (EMSOFT)*, LNCS. Springer Verlag, 2002.

[HL96] Mats Per Erik Heimdahl and Nancy G. Leveson. Completeness and consistency in hierarchical state-based requirements. *IEEE Transactions on Software Engineering*, 22(6):363–377, 1996.

[HLN+90] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark B. Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, 1990.

[HO05] David W. Humphrey and Larry O'Brien. Siemens process industry strategies. Technical report, ARC Advisory Group, Dedham, MA 02026 USA, Jan 2005.

[Hoo02] Jozef Hooman. Towards formal support for uml-based development of embedded systems. In *Proceedings of the 3d PROGRESS Workshop on Embedded Systems*, pages 71–76, Utrecht, 2002.

[HTBSL78] Albert L. Hopkins, III T. Basil Smith, and Jaynarayan H. Lala. Ftmp - a highly reliable fault-tolerant multiprocess for aircraft. *Proceedings of the IEEE*, 66(10):1221–1239, Oct 1978.

[IC02] Magnus Larsson Ivica Crnkovic, editor. *Building Reliable Component-Based Software Systems*. Artech House, 2002.

[iee00] Ieee recommended practice for architectural description of software intensive systems. Technical report, IEEE, 2000.

[IF99] Damir Isovic and Gerhard Fohler. Handling sporadic tasks in off-line scheduled distributed real-time systems. In *Proceedings of the 11th International EUROMICRO Conference on Real-Time Systems (ECRTS 99)*, 1999.

[Ins08] Institute for Software Integrated Systems. Metagme -gme metamodeling environment. http://www.isis.vanderbilt.edu/projects/gme/meta.html,

2008.

[Int98] International Electrotechnical Commission. *IEC 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems*, 1998.

[JSEB04] Ian Johnson, Colin F. Snook, Andy Edmunds, and Michael Butler. Rigorous development of reusable, domain-specific components, for complex applications. In *CSDUML'04 - 3rd International Workshop on Critical Systems Development with UML*, 2004.

[KB03] Hermann Kopetz and Günther Bauer. The Time-Triggered Architecture. *Proceedings of the IEEE*, 91(1):112 – 126, January 2003.

[KBWI99] Zbigniew Kalbarczyk, Saurabh Bagchi, Keith Whisnant, and Rafishankar K. Iyer. Chameleon: A software infrastructure for adaptive fault tolerance. *IEEE Transactions on Parallel and Distributed Systems*, pages 382–401, Jun 1999.

[KGG+06] Mohammad Ullah Khan, Kurt Geihs, Felix Gutbrodt, Peter Gohner, and Roland Trauter. Model-driven development of real-time systems with uml 2.0 and c. In *Proceedings of the Fourth Workshop on Model-Based Development of Computer-Based Systems and Third International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MBD-MOMPES'06)*, pages 33–42, Washington, DC, USA, 2006. IEEE Computer Society.

[KGR91] Hermann Kopetz, Günther Grünsteidl, and Johannes Reisinger. Fault-tolerant membership service in a synchronous distributed real-time system. In *Dependable Computing for Critical Applications*, pages 411–429, 1991.

[Kir02] Christoph Meyer Kirsch. Principles of real-time programming. In *EMSOFT '02: Proceedings of the Second International Conference on Embedded Software*, pages 61–75, London, UK, 2002. Springer-Verlag.

[Kop97] Hermann Kopetz. *Real-Time Systems, Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Dordrecht, Nederlands, 1997.

[Kop06] Hermann Kopetz. On the fault hypothesis for a safety-critical real-time system. In *Automotive Software - Connected Services in Mobile Networks*, volume 4147 of *Lecture Notes in Computer Science*, pages 31–42, Heidelberg, Oct 2006. Springer.

[KSH05] Christoph M. Kirsch, Marco A. Sanvido, and Thomas A. Henzinger. A programmable microkernel for real-time systems. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 35–45, New York, NY, USA, 2005. ACM.

[KSLB03] Gabor Karsai, Janos Sztipanovits, Akos Ledeczi, and Ted Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1):145–164, 2003.

[Kul99] Sandeep S. Kulkarni. *Component based design of fault-tolerance*. PhD thesis, Ohio State University, 1999. Adviser-Anish Arora.

[LA90] Pete A. Lee and Tom Anderson. *Fault Tolerance: Principles and Practice*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1990.

[Lam81] Butler W. Lampson. Atomic transactions. In *Distributed Systems - Architecture and Implementation, An Advanced Course*, pages 246–265. Springer-Verlag, 1981.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[Lam02] Leslie Lamport. *Specifying Systems*. Addison-Wesley, November 2002.

[Lan66] Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.

[Lap85] Jean-Claude Laprie. Dependable computing and fault-tolerance: Concepts and terminology. In *Proceedings of the 15th International Symposion on Fault Tolerant Computing Systems*, pages 2–11, Ann Arbor, MI, USA, Jun 1985.

[Lap93] Phil Laplante. *Real-Time Systems Design And Analysis: An Engineer's Handbook*. IEEE Press, New York, 1993.

[Lee00] Edward A. Lee. What's ahead for embedded software? *Computer*, 33(9):18–26, 2000.

[Lee03] Edward A. Lee. Overview of the ptolemy project. Technical Report UCB/ERL M03/25, EECS Department, University of California, Berkeley, Jul 2003.

[Lee05] Edward A. Lee. Building unreliable systems out of reliable components: The real time story. Technical report, EECS Department, University of California, Berkeley, Oct 2005.

[Lee06] Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[Liu00] Jane W. S. W. Liu. *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[LL73] Chung Laung Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[LMB+01] Akos Ledeczi, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle, and Peter Volgyesi. The generic modeling environment. In *Workshop on Intelligent Signal Processing*, Budapest, May 2001. IEEE.

[LMS85] Leslie Lamport and Michael Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, 1985.

[LMS03] Luciano Lavagno, Grant Martin, and Bran V. Selic, editors. *UML for Real: Design of Embedded Real-Time Systems*. Springer, 2003.

[LR80] Butler W. Lampson and David D. Redell. Experience with processes and monitors in mesa. *Communications of the ACM*, 23(2):105–117, Feb 1980.

[LRT92] John P. Lehoczky and Sandra Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. In

Proceedings of the IEEE Real-Time Systems Symposium, pages 110–123, Dec 1992.

[LS93] Bev Littlewood and Lorenzo Strigini. Validation of ultrahigh dependability for software-based systems. *Communications of the ACM*, 36(11):69–80, 1993.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Language Systems*, 4(3):382–401, 1982.

[LWL88] Jennifer Lundelius-Welch and Nancy Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation*, 77(1):1–36, 1988.

[Mar06] Peter Marwedel. *Embedded System Design*. Springer, 2006.

[MEG+04] M.Marcos, E.Estévez, U. Gangoiti, I. Sarachaga, and J. Barandiarán. Uml modeling of industrial distributed control systems. In *Proceedings of the 6th Portuguese Conference on Automatic Control*, 2004.

[MEH01] Mark W. Maier, David Emery, and Rich Hilliard. Software architecture: introducing ieee standard 1471. *Computer*, 34(4):107–109, Apr 2001.

[met07] Domain-specific modeling with metaedit+: 10 times faster than uml. Technical report, MetaCase, 2007.

[MLJ03] Jean-Noel Meunier, Frank Lippert, and Ravi Jadhav. Rt modeling with uml for safety critical applications - the hidoors project example. In *Proc. Workshop on Specification and Validation of UML Models for Real-Time and Embedded Systems*, 2003.

[MM88] Aamer Mahmood and Edward J. McCluskey. Concurrent error detection using watchdog processors - a survey. *IEEE Transactions on Computers*, 37(2):160–174, Feb 1988.

[MM03] Joaquin Miller and Jishnu Mukerji. *MDA Guide*. Object Management Group, Inc., Jun 2003. Version 1.0.1 (omg/03-06-01).

[MNP95] John A. McDermid, Mark Nicholson, and David J. Pumfrey. Experience with the application of HAZOP to computer-based systems. In *10th Annual Conference on Computer Assurance*, pages 37–48, Gaithersburg, Maryland, 1995. IEEE.

[Nil04] Kelvin Nilsen. Certification requirements for safety-critical software. *RTC Magazine*, Jun 2004.

[Obj03] Object Management Group. *UML 2.0 OCL Specification*, Oct 2003.

[Obj04a] Object Management Group. *UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms*, 1.0 edition, Jun 2004.

[Obj04b] Object Management Group. *UML Profile for Schedulability, Performance, and Time Specification*, 1.1 edition, Jun 2004.

[Obj06] Object Management Group. *Meta Object Facility (MOF) Core Specification*, 2.0 edition, Jan 2006.

[Obj07a] Object Management Group. *OMG Systems Modeling Language (OMG SysML), V1.0*, 1.0 edition, Sep 2007.

[Obj07b] Object Management Group. *OMG Unified Modelling Language Specification*, 2.1.2 edition, Nov 2007.

[Obj07c] Object Management Group. *A UML Profile for MARTE, Beta 1*, Aug 2007.

[PABD+99] David Powell, Jean-Claude Arlat, Ljerka Beus-Dukic, Andrea Bondavalli, P. Coppola, A. Fantechi, E. Jenn, C. Rabejac, and Andrew Wellings. Guards: A generic upgradable architecture for real-time dependable systems. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):580–599, 1999.

[PBWB00] Stefan Poledna, Alan Burns, Andy Wellings, and Peter Barrett. Replica determinism and flexible scheduling in hard real-time dependable systems. *IEEE Transactions on Computers*, 49:100–110, February 2000.

[PCD91] David Powell, Marc Chérèque, and David Drackley. Fault-tolerance in delta-4. *ACM SIGOPS Operating Systems Review*, 25(2):122–125, Apr 1991.

[Phi03] Don Phillips. Major us airlines complete safest year. *Washington Post*, June 13th:10, 2003.

[Pol94] Stefan Poledna. Replica determinism in distributed real-time systems: A brief survey. *Real-Time Systems*, 6:289–316, May 1994.

[Pow94] David Powell. Distributed fault tolerance - lessons learned from delta-4. In *Revised Papers from a Workshop on Hardware and Software Architectures for Fault Tolerance*, pages 199–217, London, UK, 1994. Springer-Verlag.

[Pra96] Dhiraj K. Pradhan. *Fault-Tolerant Computer System Design*. Prentice Hall, 1996.

[PV00] Luís M. Pinho and Francisco Vasques. Timing analysis of reliable real-time communication in can networks. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, Delft, Netherlands, 2000.

[PVW04] Luís M. Pinho, Francisco Vasques, and Andy Wellings. Replication management in reliable real-time systems. *Real-Time Systems*, 26(3):261–296, 2004.

[Ram97] J.C. Ramming, editor. *Proceedings of the Conference on Domain-Specific Languages*. USENIX, Santa Monica, CA, USA, Oct 1997.

[Ran75] Brian Randell. System structure for software fault tolerance. In *Proceedings of the International Conference on Reliable Software*, pages 437–449, New York, NY, USA, 1975. ACM.

[RC04] Jorge Real and Alfons Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Syst.*, 26(2):161–197, 2004.

[RCV+05] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. Swift: Software implemented fault tolerance. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 243–254, San Jose, CA, 2005. IEEE Computer Society.

[Rei79] D. Reifer. Software failure modes and effects analysis. *IEEE Transactions on Reliability*, 28(3):247–249, Aug 1979.

[RLT78] Brian Randell, Pete Lee, and Phillip C. Treleaven. Reliability issues in computing system design. *ACM Computing Surveys*, 10(2):123–165, 1978.

[RM00] Eberhardt Rechtin and Mark W. Maier. *The Art of Systems Architecting*. CRC Press, Inc., 2nd edition, 2000.

[RP07] Alek Radjenovic and Richard Paige. *Architecting Dependable Systems IV*, volume 4615 of *Lecture Notes in Computer Science*, chapter The View Glue, pages 66–88. 2007.

[RSB90] Parameswaran Ramanathan, Kang G. Shin, and Ricky W. Butler. Fault-tolerant clock synchronization in distributed systems. *Computer*, 23(10):33–42, 1990.

[Rus96] John Rushby. Reconfiguration and transient recovery in state machine architectures. In *Proceedings of the 26th Annual International Symposium on Fault-Tolerant Computing (FTCS 1996)*, pages 6–15, Sendai, Japan, 1996. IEEE Computer Society.

[SAE04] SAE International. *Architecture Analysis & Design Language (Aadl)*, Nov 2004.

[Sch90] Henk Schepers. Towards a formal framework for fault-tolerance. In *EW 4: Proceedings of the 4th workshop on ACM SIGOPS European workshop*, pages 1–4, Bologna, Italy, 1990. ACM.

[SKS95] Shirish S. Sathaye, Daniel I. Katcher, and Jay K. Strosnider. Fixed priority scheduling with limited priority levels. *IEEE Transactions on Computers*, 44(9):1140–1144, 1995.

[SS83] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *Computer Systems*, 1(3):222–238, 1983.

[SS97] Ulrich Schmid and Klaus Schossmaier. Interval-based clock synchronization. *Real-Time Systems*, 12(2):173–228, 1997.

[SSBG03] Shankar Sastry, Janos Sztipanovits, Ruzena Bajcsy, and Helen Gill. Scanning the issue - special issue on modeling and design of embedded software. *Proceedings of the IEEE*, 91(1):3–10, 2003.

[SSG91] Ajit Singh, Jonathan Schaeffer, and Mark Green. A template-based approach to the generation of distributed applications using a network of workstations. *IEEE Transactions on Parallel and Distriuted Systems*, 2(1):52–67, Jan 1991.

[ST87] T. K. Srikanth and Sam Toueg. Optimal clock synchronization. *Journal of the Association for Computing Machinery*, 34(3):627–645, Jul 1987.

[Sta88] John A. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. *Computer*, 21(10):10–19, Oct 1988.

[Sur94] Neeraj Suri. *Advances in ULTRA-Dependable Distributed Systems*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1994.

[SV06] Thomas Stahl and Markus Voelter. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley, 1 edition, May 2006.

[Szy02] Clemens Szyperski. *Component Software*. Addison-Wesley Professional, nov 2002.

[TCC05] Martin Torngren, DeJiu Chen, and Ivica Crnkovic. Component-based vs. model-based development: A comparison in the context of vehicular embedded systems. In *EUROMICRO '05: Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 432–441, Washington, DC, USA, 2005. IEEE Computer Society.

[Tia95] Too-Seng Tia. *Utilizing slack time for aperiodic and sporadic requests scheduling in real-time systems*. PhD thesis, University of Illinois at Urbana-Champaign, April 1995.

[Tol04a] Juha-Pekka Tolvanen. Making model-based code generation work. *EmbeddedSystems Europe*, pages 36–38, Aug 2004.

[Tol04b] Juha-Pekka Tolvanen. Metaedit+: Domain-specific modeling for full code generation demonstrated. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 39–40, Vancouver, Oct 2004.

[TTT03] TTTech Computertechnik AG. *Time Triggered Protocol TTP/C High-Level Specification Document*, 2003.

[TTT05] TTTech Computertechnik AG. *TTPOS: The OSEKtime-Based Operating System for Safety-Critical Real-Time Applications*, 2005.

[VC02] Paulo Veríssimo and António Casimiro. The timely computing base model and architecture. *IEEE Transactions on Computers*, 51(8):916–930, 2002.

[VCP+00] Paulo Veríssimo, António Casimiro, Luís M. Pinho, Francisco Vasques, Luís Rodrigues, and Eduardo Tovar. Distributed computer-controlled systems: the DEAR-COTS approach. In *Proceedings of the 16th IFAC Workshop on Distributed Computer Control Systems*, pages 128–125, Sydney, Australia, Nov 2000. Elsevier Science Ltd.

[VGRH81] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl. *Fault Tree Handbook*. U.S. Nuclear Regulatory Commission, 1981.

[Vog88] Udo Voges, editor. *Software diversity in computerized control systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1988.

[VSK05] Markus Voelter, Christian Salzmann, and Michael Kircher. Model driven software development in the context of embedded component infrastructures. In *Component-Based Software Development for Embedded Systems*, number 3778 in Lecture Notes in Computer Science, pages 143–163. Springer, 2005.

[VSWH02] Joao Ventura, Fridtjof Siebert, Andy Walter, and James Hunt. Hidoors-a high integrity distributed deterministic java environment. In *Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'02)*, pages 113–118, Los Alamitos, CA, USA, 2002. IEEE Computer Society.

[Web88] D. G. Weber. Specifications for fault-tolerance. Technical Report 19-3,

Odyssey Research Associates, New York, 1988.

[WH99a] Michael W. Whalen and Mats Per Erik Heimdahl. An approach to automatic code generation for safety-critical systems. In *Proceedings of the 14th IEEE International Conference on Automated Software Engineering*, pages 315–318, Oct 1999.

[WH99b] Michael W. Whalen and Mats Per Erik Heimdahl. On the requirements of high-integrity code generation. In *Proceedings of the Fourth High Assurance in Systems Engineering Workshop*, pages 217–224, Nov 1999.

[Wil03] Reinhard Wilhelm. Run-time guarantees for real-time systems. *Formal Modeling and Analysis of Timed Systems, First International Workshop, FORMATS 2003, Revised Papers*, LNCS 2791:166–167, 2003.

[WLG+78] John H. Wensley, Leslie Lamport, J. Goldberg, M.W. Green, K.N. Levitt, Michael Melliar-Smith, Robert Shostak, and Chuck Weinstock. Sift: Design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE*, 66(10):1240–1255, oct 1978.

[ZBL07] Gregory Zoughbi, Lionel Briand, and Yvan Labiche. A uml profile for developing airworthiness-compliant (rtca do-178b), safety-critical software. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007, Nashville, USA, September 30 - October 5, 2007, Proceedings*, volume 4735 of *Lecture Notes in Computer Science*, pages 574–588, Sep 2007.