FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Simulated Fault Injection for Time-Triggered Safety-Critical Embedded Systems

DISSERTATION

zur Erlangung des akademischen Grades

Doktor/in der technischen Wissenschaften

eingereicht von

Iban Ayestaran Cipitria
Matrikelnummer 1228329

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Peter Puschner

Diese Dissertation haben begutachtet:

(Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Peter Puschner)

(Prof. Dipl.-Ing. Dr. Gerhard Fohler)

Wien, TT.MM.JJJJ

(Iban Ayestaran Cipitria)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Simulated Fault Injection for Time-Triggered Safety-Critical Embedded Systems

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor/in der technischen Wissenschaften

by

Iban Ayestaran Cipitria
Registration Number 1228329

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Peter Puschner

The dissertation has been reviewed by:

(Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Peter Puschner)	(Prof. Dipl.-Ing. Dr. Gerhard Fohler)

Wien, TT.MM.JJJJ

(Iban Ayestaran Cipitria)

Erklärung zur Verfassung der Arbeit

Iban Ayestaran Cipitria
Bidebieta Auzoa 1A, Aiestaran Enea. 20400 Tolosa

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Acknowledgements

This dissertation has been carried out in collaboration between the Electronics department of IK4-Ikerlan Research Center (Basque Country, Spain), and the Cyber-Physical Systems group of Vienna University of Technology (TU Wien). I would like to give thanks to all the people that contributed to its development.

First of all, I would like to thank my supervisor, Professor Peter Puschner from TU Wien, for his support during these years and his involvement in the supervision, including his trips to the Basque Country. Thanks also to Professor Gerhard Fohler from TU Kaiserslautern for his work as co-reviewer of this thesis.

Many thanks to Dr. Jon Perez, head of the Electronics area in IK4-Ikerlan, for his guidance, encouragement and full support. To Carlos Fernando Nicolas for all his advice, ideas, criticism and motivation that really made this thesis possible. To Christian El Salloum for his supervision during the first year of my work.

Many thanks to IK4-Ikerlan for giving me the opportunity to carry out a PhD in collaboration with TU Wien. I would like to mention all my colleagues in Ikerlan for their friendship and good atmosphere, and specially those that contributed somehow to my thesis, Gaizka, Mikel, Irune and Jan. Thanks also to the Cyber-Physical systems group of TU Wien for their hospitality, particularly to Haris, Bekim, Benedikt, Roland and Armin.

And finally, very special thanks to all my family, specially my parents Axun and Jose Luis and my sister Nerea for their comprehension, confidence, their optimism even in the difficult moments, and all the opportunities they gave me. To my girlfriend Stephanie, for her full support and motivation. To Pello, Kristina and Irati, for their valuable advice. And to my friends, Txorimaloak, for our interesting discussions in any other topic aside from work.

To all of you, eskerrik asko.

Abstract

Safety-critical systems are dependable systems that could lead to loss of life, significant property damages or damages to the environment in case of failure, such as avionic and railway systems, offshore mills or nuclear power plants. Systems of this type must satisfy strict temporal constraints in order to guarantee certain safety properties. Besides, these systems must provide a certain degree of fault-tolerance, to guarantee that they keep a safe behavior even in the presence of faults in the system.

Historically, the control of safety-critical systems used to be handled by mechanical devices. However, due to the wide spectrum of possibilities that computer systems offer, these systems are nowadays commanded by computers. The most significant progress in this field may be the fly-by-wire system developed for the latest Airbus A380, which is fully controlled by a computer system. Moreover, the increasing functionality demanded by industry has lead to a considerable complexity growth. For example, high-end cars had about 70 electronic control units (ECUs) back in 2006 , and this number raised up to 100 by the year 2011.

Therefore, tackling the complexity challenge and preserving time properties and constraints throughout the development process are key challenges in the field. With this goal, this research work presents the Platform Specific Time-Triggered Model (PS-TTM), a novel model-based development framework based on SystemC for time-triggered safety-critical embedded systems. The proposed modeling work-flow tackles the complexity challenge following the MDA process and the Y-chart paradigm, by raising the level of abstraction at the very first development stages and creating a purely functional Platform Independent Model (PIM). Once this abstract model is validated, HW-related concepts are integrated into the model and the Platform Specific Model is generated.

The work includes the PS-TTM Automatic Test Executor (PS-TTM ATE), a time-triggered testing and simulated fault injection framework for the validation of both platform independent and platform specific models of systems. The PS-TTM ATE provides a simulation environment that enables the test developers to check the behavior of the system under the considered circumstances. Besides, PS-TTM ATE includes a non-intrusive fault injection mechanism that allows testing teams to inject faults in the models during simulation, in order to evaluate the effectiveness of the fault-tolerance mechanisms implemented in them before assembling a system prototype.

Kurzfassung

Sicherheitskritische Systeme sind Systeme, die im Fall einer Fehlfunktion zu tödlichen Unfällen, schwerwiegenden Eigentumsverlusten oder gravierenden Umweltschäden führen können. Beispiele für sicherheitskritische Systeme finden sich z.B. in der Luftfahrt, im Zugverkehr, in Offshore Windanlagen und Kernkraftwerken. Sicherheitskritische Systeme müssen strenge zeitliche Anforderungen erfüllen, um in jedem Moment die Sicherheit des Systems zu garantieren. Zusätzlich müssen sicherheitskritische Systeme einen gewissen Grad an Fehlertoleranz aufweisen, um auch im Falle eines Fehlers die Sicherheit des gesamten Systems gewährleisten zu können.

Früher basierten sicherheitskritische Systeme hauptsächlich auf mechanischen Elementen, während heutzutage mehr und mehr rein computergestützte Systeme eingesetzt werden. Einer der wichtigsten Fortschritte im Bereich rein computergestützter Systeme war sicherlich das Fly-by-Wire System des neuen Airbus A380, dessen Steuerungssystem vollständig computerbasiert ist. Zusätzlich zum Einsatz von computergestützten Systemen führt die zunehmende Funktionalität, die von der Industrie gefordert wird, zu einer beträchtlichen Steigerung der Komplexität aktueller sicherheitskritischer Systeme. Ein Auto der Oberklasse hatte zum Beispiel im Jahr 2006 um die 70 elektronische Steuergeräte, während diese Zahl im Jahr 2011 auf über 100 gestiegen ist.

Die Beherrschung der gestiegenen Komplexität bei gleichzeitiger Einhaltung aller zeitlichen Anforderungen ist eine zentrale Herausforderung für aktuelle sicherheitskritische Systeme. Mit dem Ziel, diese Herausforderung anzugehen präsentiert die vorliegende Forschungsarbeit das *Platform Specific Time-Triggered Model (PS-TTM)*, ein neuartiges modellbasiertes Entwicklungs-Framework in SystemC für zeitgesteuerte, sicherheitskritische Systeme. Zusätzlich zum Framework definiert die Forschungsarbeit einen neuartigen Workflow, der auf dem MDA-Process und dem Y-Chart Entwicklungsparadigma basiert. Der vorgeschlagene Workflow erlaubt die Beherrschung der Komplexität komplexer sicherheitskritischer Systeme dadurch, dass Systeme in den ersten Entwicklungsphasen mit plattformnabhängigen Modellen (PIMs) auf einem hohen Abstraktionsniveau modelliert werden. Nach der Validierung dieser abstrakten, plattformnabhängigen Modelle werden die Modelle um Hardware-spezifische Eigenschaften angereichert, um die plattformpezifischen Modele (PSMs) des Systems zu erhalten.

Neben dem Entwicklungs-Framework und dem Workflow beinhaltet die Forschungsarbeit den PS-TTM Automatic Test Executer (PS-TTM ATE), ein zeitgesteuertes Test- und Fehlerinjizierungsframework zur Validierung von plattformunabhängigen und plattformspezifischen Modellen. Das PS-TTM ATE Framework stellt eine Simulationsumgebung zur Verfügung, mit der der Testentwickler das Verhalten des Systems in verschiedenen Situationen simulieren und validieren kann. Zusätzlich unterstützt PS-TTM ATE die nicht intrusive Fehlerinjizierung, welche es erlaubt, während der Simulation Fehler in die Modelle zu einzustreuen, um damit die Effektivität der im Modell implementierten Fehlertoleranzmechanismen zu untersuchen.

Laburpena

Segurtasun kritikoko sistemak, akatsak jasanez gero kalte ekonomiko, ekologiko edo humanitario larriak eragin ditzaketen sistema txertatuak dira, sistema aeroespazialak, trenbide sareak, sistema eolikoak edota nuklearrrak esate baterako. Sistema hauek ezaugarri tenporal zehatzak bete behar izaten dituzte segurtasun maila jakin bat bermatu ahal izateko. Gainera, sistema hauek hutsegiteak modu jakinean jasateko gaitasuna izan behar dute, akats larrienak gertatuz gero ere jokaera segurua mantendu dezaten.

Duela urte gutxi arte, segurtasun kritikoko sistemen kontrola gailu mekanikoen ardura izan ohi zen. Azken hamarkadetatik hona, ordea, geroz eta ohikoagoa da sistema informatikoak erabiltzea sistema mota hauek kontrolatzeko. Adibide bezala, alor honetan lortu den aurrerapen nagusienetariko bat Airbus A380 hegazkinarentzat eraikitako *Fly-by-wire* sistema da. Sistema hauen berezko konplexutasuna handia izanik, gaur egun zailtasun hau areagotzen ari da, industriak eskatzen dituen funtzio gehigarrien ondorioz. Esaterako, 2006.urtean, luxuzko auto batek 70 kontrol elektronikoko unitate (ECU) inuguru zituen; 5 urte beranduago, 2011n, ECU kopurua 100etik gorakoa zen. Arrazoi hauengatik, gaur egungo segurtasun kritikoko sistemen garapenaren konplexutasun maila txikiagotzea, euren denbora-ezaugarriak murriztu gabe, erronka itzela da.

Helburu honekin, ikerketa lan honek Platform Specific Time-Triggered Model (PS-TTM) izeneko plataforma aurkezten du, denboraz jaurtiriko segurtasun kritikoko sistemak SystemC lengoaian garatzeko eta aztertzeko erreminta. Sistemen garapena errazteko asmoz, tesi honetan proposaturiko lan organigramak MDA eta Y-chart metodologiak jarraitzen ditu, garapen prozesuaren lehen etapetan modeloen abstrakzio maila igoz eta modelo funtzional soilak sortuz. Modelo funtzional abstraktu hauek hobetsita daudenean, hardware-arekin zerikusia duten kontzeptuak gehitzen zaizkie modelo horiei, abstrakzio maila modu kontrolatuan murriztuz.

Honetaz gain, lan honek PS-TTM Automatic Test Executor (ATE) deituriko plataforma aurkezten du. Plataforma hau PS-TTM bidez diseinaturiko sistema txertatuen jokabidea egiaztatzea ahalbidetzen duen test ingurune bat da. PS-TTM ATE-ak abstrakzio maila desberdinetan definituriko modeloetan akatsak txertatzeko eta simulatzeko gaitasuna du. Honi esker, diseinatzaileek euren sistemek akatsen aurrean erakusten duten erreakzioa aztertu eta akatsen aurkako mekanismoen eraginkortasuna ebaluatu dezakete, sistemen prototipo fisikoak eskuragarri izan aurretik.

Contents

List of Figures

List of Tables

List of Listings

CHAPTER 1

Introduction

1.1 Motivation

An embedded system is a system that uses processors and special hardware for dedicated control functions, and interacts with a real-life environment [TLMP93, ZN08]. The capability of the semiconductor industry to reduce the size of integrated components has lead to a big increase in the number of components integrated in a specific silicon area, and follows the well-known Moore's law [Moo65]. This has enabled an augmentation in the functionality of embedded systems but has also prompted a considerable growth in their complexity.

Safety-critical embedded systems are dependable systems that could lead to loss of life, significant property damages or damages to the environment in case of failure. Therefore, safety-critical systems must satisfy a certain degree of fault-tolerance, as required by the safety standards. These systems have also suffered a complexity increase in the last years due to the new functionalities included in modern embedded systems. For example, high-end cars had about 70 electronic control units (ECUs) back in 2006 [Bro06], and this number raised up to 100 in the year 2011 [BCR^{+}11]. This implies not only an increment on the cost and complexity of the systems, but also a growth of the amount of potential defects [EJ09]. Moreover, as Di Natale et al. point out in [DNSV10], it is expected that factors like the interdependency of functions and the cost of each ECU will lead to a transition from traditional federated architectures to integrated architectures where one ECU supports multiple functions, which may cause a growth in the complexity of ECUs.

Therefore, tackling the complexity challenge [Kop08] while providing a consistent notion of time and preserving properties through the development process [HS07, JTM07] is a key challenge in the field [Per11]. In this context, the well known Y-chart development process [BCG^{+}97, KDVvdW97] specifies the platform model and the functional model of the system separately, and combines both by means of a mapping model to obtain a model of a complete system. Moreover, Model Driven Development (MDD) approaches, such as the Model Driven Architecture (MDA) [MM03], are also based on the idea of a strict separation of behavioral and platform models. More specifically, MDA proposes the development of a Platform Specific Model (PSM) by applying transformations to a Platform Independent Model (PIM).

Since failures in safety-critical systems may cause human, environmental and economical damages, the dependability assessment plays a crucial role in their development. The correctness of system functions is usually addressed by formal reasoning. For example, relying on a formally specified Model of Computation (MoC), such as the time-triggered architecture (TTA), the synchronous reactive MoC (SR), or the logical execution time MoC (LET), enables reasoning about the temporal behavior of functional systems. However, when an exhaustive proof of correctness cannot be achieved, the modeling language should support simulation and testing [DNSV10].

Nowadays, it is estimated that verification and validation activities take approximately 50% of the total development effort of an embedded system [Enc03, Kop11]. For safety-critical systems, this percentage is even higher [Kop11]. Besides, the cost of finding and fixing defects increases exponentially as the development process progresses. In the worst case, [DES01] calculates that calling back a car that suffers from a safety-critical failure can imply a higher cost than the cost of its detailed testing activities. Therefore, advances on frameworks and methodologies for simulation and testing of system models at the early stages of development may provide a positive impact on the development of safety-critical embedded systems. In fact, a recent survey about embedded system design projects [Ham14] confirms that 63% of the embedded system developers consider simulation the most important verification technique for their future developments.

The complexity of modern safety-critical systems, the high integration in silicon components and the limitations of software and hardware technologies force safety-critical systems to coexist with faults. This problem is addressed by the introduction of fault-tolerance mechanisms (FTMs) into the systems, as safety-standards recommend [IEC10, ISO09, CEN11]. In order to assess their effectiveness and so to evaluate the dependability of the system, fault-tolerance mechanisms need to be exercised by means of fault injection techniques.

Among the different fault injection approaches [BP03, HTI97], simulated fault-injection (SFI) enables performing this fault-tolerance assessment from the early stages of the design, therefore reducing the risk of a late and expensive detection of safety pitfalls. However, time-triggered modeling approaches and frameworks such as the Executable Time-Triggered Model (E-TTM) [PNOES10a, Per11] or Timing Definition Language (TDL) [Chr14] do not natively offer a framework for dependability assessment by simulated fault injection. Therefore, fault injection is usually carried out by manually modifying the system to insert the desired fault injection mechanisms in it [PAaP10]. This presents two important drawbacks. First, a new model needs to be created and compiled for each fault to be tested, which increases validation and verification costs. Second, since a new model is created and validated for each faulty case, in the end, the original model is never validated against faults, but only derived models are. This leaves a degree of uncertainty on the behavior of the original model against faults, since the designers might involuntarily change some functional property when creating the modified model.

Given this situation, this dissertation presents the Platform Specific Time-Triggered Model (PS-TTM), a novel modeling and simulation framework for the design and assessment of safety-critical time-triggered systems according to the MDA and the Y-chart development process. As suggested by these approaches, the design of systems in PS-TTM begins with the description of a purely functional model of the system. To do so, the herein presented framework includes the Platform Independent Time-Triggered Model (PI-TTM), a design and simulation environment for functional models based on the LET MoC. The PI-TTM has been built on top of the PS-TTM framework, in a way that it provides a seamless connection between platform independent and platform specific models.

Our simulation engine for PI-TTM and PS-TTM models enables the validation of fault-tolerance mechanisms by non-intrusive simulated fault injection from the early stages of the design, with a mechanism that prevents the designers from generating a new model for each fault to be injected, thus reducing the time and effort of validation activities and increasing the level of confidence on the results obtained in the tests. The fault injection campaigns can be applied to models at different abstraction levels, and are carried out by the time-triggered testing and fault injection framework included in this work, called the PS-TTM Automatic Test Executor (ATE).

Although different models of computation, languages and frameworks can be found in the state of the art for the development of safety-critical embedded systems (see chapters 2 and 3), the systematic preservation of time properties throughout the model down to the implementation is still a challenge. In this context, the Time-Triggered Architecture (TTA) provides a validated and certifiable core technology for the development of safety-critical embedded systems [JSPP04], based on the time-triggered MoC. To take advantage of this, the PS-TTM approach relies on the Executable Time-Triggered Model (E-TTM) [PNOES10a].

Why E-TTM?

The time-triggered MoC guarantees that time properties and constraints are intrinsically preserved down to the final implementation when the system is based on the TTA [Per11], what makes it an appropriate candidate for the design of safety-critical systems. Unlike other time-triggered modeling approaches, such as TDL [Chr14] or TMO

[Kim97], which present some differences to the time-triggered MoC that limit their applicability [Per11], E-TTM provides a closer approximation to the time-triggered MoC.

Moreover, the E-TTM meta-model is developed in SystemC, which has become the de-facto standard for HW/SW system modeling. In addition to this, E-TTM provides different techniques such as abstraction, partitioning and segmentation to tackle the complexity challenge of modern embedded systems.

However, the E-TTM may be found too specific for the modeling of purely functional models of systems. For example, it requires the specification of a detailed time-triggered schedule, which may not be interesting at the earliest development stages. In that case, languages based on the synchronous reactive MoC (e.g., Lustre [HCRP91], Esterel [Ber98]) or on logical execution time (TDL [Chr14], HTL [GSVK$^+$06]) give a higher abstraction to the designer, which makes them more suitable for the development of PIMs.

Hence, this approach relies on the LET MoC for the development of platform independent models, whilst platform specific models are developed relying on E-TTM.

Why LET?

Synchronous languages [BCE$^+$03], such as Lustre [HCRP91, Hal05], Esterel [Ber98, Ber00] or the commercial tool SCADE Suite [wwww], are based on the SR MoC. The SR MoC relies on the synchronous hypothesis, which defines an abstraction where the execution of jobs and communication of data are considered instantaneous and simultaneously triggered. This way the notion of time is abstracted from the models so that the models can be understood as mathematical equations, which eases the formal verification of their properties. For this reason, synchronous languages are used for the design of functional models of dependable and safety-critical embedded systems.

However, the implementation of SR models requires matching the synchrony paradigm with the system architectures and the environment, which may be invalidated by the dynamics of the environment. Moreover, the deployment of SR models on distributed platforms that do not provide strict time determinism, or the implementation of interfaces for analog components (sensors or actuators) becomes cumbersome, in spite of the advances in research [DNSV10].

Therefore, in order to ease the design of platform independent models, PI-TTM relies on the LET MoC [KS12]. In the LET MoC, described in detail in section 2.5, the execution of functions takes a fixed logical duration regardless of their physical duration, whereas communication between components is instantaneous and is only triggered at the logical start and end points of jobs. This way, the LET MoC can be considered an abstraction of the Time-Triggered MoC.

The LET MoC has been adopted by a number of different approaches and languages, such as the Timing Definition Language (TDL) [RDPN10] and the Hierarchical Timing Language (HTL) [GSVK$^+$06]. However, in order to ease the transition from the PIM to the PSM, this thesis presents a new LET engine built as an abstraction of the E-TTM engine, which enables modeling LET based functional systems in SystemC. This way, the approach described in this dissertation provides a seamless connection between the LET based and E-TTM based models.

1.2 Goals of the thesis

The main goal of this thesis is the "definition of an executable time-triggered safety-critical systems-modeling approach based on the Y-chart development process and the MDA, and the development of a testing framework which enables non-intrusive simulated fault injection at the different stages of the development for the verification and validation of such systems".

In order to tackle the challenges identified in the previous section, the modeling and simulation approach should meet the following characteristics:

G1. **Time and value domain determinism**: Safety-critical systems must enable certification in accordance to applicable safety standards. Determinism eases the certification process by reducing the state space of systems.

G2. **Support strategies to tackle the complexity challenge**: As described in section 1.1, complexity is one of the biggest concerns in the embedded systems field. Hence, the modeling approach must tackle this challenge by enabling different techniques identified in the state of the art, such as abstraction, partitioning and segmentation.

G3. **Be compliant with the MDA approach**: MDA suggests separating the specification of the functionality of the system from the details of the target platform in order to reduce complexity and increase portability and re-usability. The approach developed within this thesis should comply with the MDA.

G4. **Allow extendability and specialization of platform components**: As technology advances, new platforms and devices are developed. In order to support new components, the platform component library must be extendable. Moreover, platform component definitions must be generic and allow specialization to enable modeling at different abstraction levels.

On the other hand, the testing and simulated-fault injection framework should:

G5. **Be compliant with the modeling and simulation approach**: The main purpose of the testing and simulated fault injection framework is to enable the verification and validation of systems developed following our approach. Hence, the testing and simulated fault injection framework should avoid any incompatibilities with the modeling framework, and thus provide a suitable environment to test the models developed following the modeling approach presented in this thesis against different test-cases and fault-configurations.

G6. Enable non-intrusive fault injection: Non-intrusive fault injection techniques are the ones that completely mask their presence, so that they have no effect on the system behavior apart from the faults they inject. Since the model is not modified at all to get the faults injected, the results provided by these techniques are more reliable than the results provided by intrusive mechanisms. Therefore, this framework must enable non-intrusive fault injection.

G7. Permit testing and simulated fault injection at all the stages of the development process (PIM and PSM models): As mentioned in section 1.1, fixing design pitfalls detected in the latest steps of the development requires a bigger effort. Hence, in order to enable the early detection of design flaws, testing and fault injection must be possible from the earliest stages of the design.

G8. Provide repeatability of test cases and fault injection activities: In order to confirm that a bug in the system has been successfully fixed, it is important to repeat the tests or SFI activities that exhibited the faulty behavior of the system. To that end, test cases and SFI must be repeatable. The value- and time-domain determinism and timing synchronization between the model and the testing framework are key properties for this purpose.

G9. Provide a high level of observability: The degree of observability of the internal signals of a system is a key property for the correct comprehension of its behavior, and certainly affects the time needed to locate and fix design bugs. Therefore, it would be highly advantageous to provide a high level of observability in order to reduce the verification and validation costs.

1.3 Contribution

More specifically, the contributions provided by this thesis to the state of the art are the following:

- **Definition of the PS-TTM**, a modeling and simulation framework for time-triggered safety-critical embedded systems based on the Y-chart development process and the MDA approach (goal G3), over SystemC. The PS-TTM is time and value domain deterministic (goal G1), and supports different techniques to reduce the cognitive complexity of systems (goal G2). Taking advantage of its abstraction capabilities, the PS-TTM includes a library of generic platform components, which can be further specialized by the designers (goal G4).

- **Creation of the PI-TTM library**, an extension to SystemC to support the definition and simulation of Logical Execution Time based functional systems. This library enables developers to design the functionality of their systems in a separate model, thus reducing the complexity, as suggested by the MDA (goal G3).

- **Definition of the PS-TTM ATE**, a testing and simulated fault injection framework for PS-TTM and PI-TTM models (goals G5, G7), which enables non-intrusive fault injection on all the stages of the system development process without the need to make any modification to the system models (goal G6). The PS-TTM ATE is synchronized with the simulation engines of the PS-TTM, thus guaranteeing repeatability of test cases (goal G8), and provides mechanisms to observe the values of all the internal signals of the systems (goal G9).

- **Provision of a seamless connection between the LET and E-TTM computation engines**, which enables the simulation, testing and fault injection for systems containing components at different stages of development. This feature contributes to increase the re-usability of models, as suggested by the MDA (goal G3).

- **Development of an extendable executable fault library**, with faults collected from the state of the art, in order to verify and validate the behavior of systems under fault conditions. Providing this pre-defined fault library to test engineers reduces the verification and validation costs and increases the repeatability of fault injection activities (goals G7, G8).

1.4 Structure of the thesis

This thesis is structured as described below:

- **Chapter 2:** This chapter gives an overview of the background and the basic concepts on which the work of this thesis is based. In particular, it focuses on explaining concepts regarding dependability and complexity, provides an overview of fault injection techniques, makes a review of model-based design (MBD) and gives an overview of different models of computation, mainly focusing on the time-triggered architecture.

- **Chapter 3:** This chapter analyzes the state of the art in modeling languages for safety-critical embedded systems and their simulation frameworks, and discusses different approaches in simulated fault injection on different modeling languages.

- **Chapter 4:** We describe in detail the PS-TTM, analyzing its meta-model, timing-behavior, syntax and semantics, and its PI-TTM abstraction level for functional models. We also explain how PS-TTM is mapped to the MDA, and we analyze its capability to simulate mixed-abstraction level systems.

- **Chapter 5:** The testing and simulated fault injection framework is introduced in this chapter, describing its sub-components, fault libraries for PI-TTM and PS-TTM, syntax and time synchronization.
- **Chapter 6:** This chapter introduces the set of different tools developed during the completion of this thesis and describes their integration in the overall work-flow proposed in Chapter 4.
- **Chapter 7:** This chapter describes the case study made for the evaluation of the approach and discusses the results obtained in the simulations.
- **Chapter 8:** The thesis is concluded by summing up the main conclusions and suggesting possible future work.

CHAPTER 2

Background and Basic Concepts

This chapter analyzes the background on which this thesis is based, and explains the most fundamental concepts in the fields of dependable embedded systems and the Model Based Design (MBD).

2.1 Dependability

The term dependability is widely used in the domain of embedded systems. However, the definition of the term has evolved through the years, as the following list shows:

- "Quality of the delivered service such that reliance can justifiably be placed on this service". [Lap85]
- "The property of a computer system such that reliance can justifiably be placed on the service it delivers" [LAK92]
- "The ability to deliver service that can justifiably be trusted" [ALR01, JTM07]
- "The ability of a system to avoid service failures that are more frequent and more severe than is acceptable" [ALRL04]

The paper written by Avizienis et al. [ALRL04] is usually taken by the embedded systems community as the reference for the definition of fundamental concepts regarding dependable and secure systems. Therefore, according to [ALRL04], dependability is considered an integrating concept that encompasses the following attributes:

- **Availability**: Readiness for correct service.
- **Reliability**: Continuity of correct service.

$$\cdots \longrightarrow \textbf{Fault} \xrightarrow{\textit{activation}} \textbf{Error} \xrightarrow{\textit{propagation}} \textbf{Failure} \xrightarrow{\textit{causation}} \textbf{Fault} \longrightarrow \cdots$$

Figure 2.1: Causality chain of dependability threats (from [ALRL04]).

- **Safety**: Absence of catastrophic consequences for the user(s) and the environment.
- **Integrity**: Absence of improper system alterations.
- **Maintainability**: Ability to undergo modifications and reparations.

Threats of Dependability

According to Avizienis [ALRL04], the dependability of a system has three potential threats: failures, errors and faults. Benso et al. define these three components in [BP03] as follows:

- **Fault**: Physical defect, imperfection or flaw that occurs within some hardware or software component.
- **Error**: Deviation from accuracy or correctness.
- **Failure**: The non-performance or incorrect performance of some expected action.

Errors are considered manifestations of faults. Therefore, faults are classified as *active faults* or *dormant faults*, depending on their effect. A fault is active when it produces an error; otherwise, it is dormant. An active fault is either an internal fault (fault originated inside the system boundaries) that was previously dormant and that has been activated by the computation process or environmental conditions, or an external fault. Most internal faults keep switching between active and dormant states.

Errors caused by active faults may propagate inside the component due to the computation process, transforming themselves into other errors. This phenomenon is called *internal propagation*. When an error propagates internally to the service interface of the component, the error propagates to other systems connected to that interface. This is known as *external propagation*.

Failures occur when an error internally propagates to the service interface and causes a deviation of the provided service from the expected service. In that case, the failure in a system behaves as a fault for the subsequent systems.

This relationship between faults, errors and failures is known as the causality chain of dependability threats, which is depicted in Figure 2.1.

Means of Dependability

Over the years the community has developed different means to achieve each of the attributes of dependability. Those means are grouped into four major categories [ALRL04]:

- **Fault prevention**: Prevention of the occurrence or introduction of faults.
- **Fault removal**: Reduction of the number and severity of faults.
- **Fault forecasting**: Estimation of the present number, the future incidence, and the likely consequences of faults.
- **Fault-tolerance**: Avoidance of service failures in the presence of faults.

Determinism

Determinism has been widely explored by philosophers over the years. From a philosophical point of view, the world is deterministic if and only if, given a specified way things are at time *t*, the way things go thereafter is fixed as a matter of natural law [Hoe05].

In the field of computer science, a model is considered deterministic if and only if given a set of initial conditions at a specific instant, and a sequence of future timed inputs, the outputs at any future instant are entailed and are not influenced by randomness [Kop08].

Therefore, deterministic models enable to make reliable predictions about their future state. In fact, determinism is a sufficient condition for logical reasoning. This eases the analysis of the behavior of systems, and hence facilitates their validation. In contrast, non-deterministic models are more difficult to validate, since repeated test cases do not necessarily produce identical results. Thus, whenever possible, dependable and safety-critical systems should be built as devices that can be modeled deterministically.

Reliability and Safety Analysis

Different techniques have been developed to evaluate the dependability of systems. Since one of the most important characteristics of dependable systems is their dynamic behavior, nowadays the most interesting techniques for dependability analysis focus on the assessment of reliability and safety [BKSZN10].

The objective of this analysis is to qualitatively identify the types of system failures that can occur, or to quantitatively define the distribution of the times-to-failure of a component or system [BKSZN10]. The following sections briefly describe the most widely used reliability and safety analysis techniques.

Failure Mode and Effect Analysis (FMEA)

The design of safety-critical systems starts with the safety analysis such as fault tree analysis and/or failure mode and effect analysis (FMEA) of the envisioned application [Kop11]. In fact, the IEC 61508 safety standard (Functional safety of electrical/-electronic/programmable electronic safety-related systems) [IEC10] recommends the FMEA as one way to assess the system reliability.

The FMEA (Failure Mode and Effect Analysis) is a systematic method of identifying and preventing product and process problems before they occur. FMEAs are focused on preventing defects, enhancing safety, and increasing customer satisfaction [MMB09]. The aim of the FMEA is to identify potential faults of the system and determine the resulting error effects before the faults occur, in order to prevent safety accidents and incidents.

The typical FMEA consists of the development of a document that follows a bottom-up flow in the analysis of the system. The technique starts with the analysis of possible failure modes of components within the system, and continues upwards until the failure modes of the whole system are evaluated. The evaluation of the safety hazards is subjective, since it is based on the previous knowledge of the designers [RPV$^+$13]. Therefore, the FMEA requires a team of experienced engineers to identify all possible failure modes of each component.

The FMEA process finishes with the development of the FMEA document, which is written in a specific FMEA worksheet. This document collects all the important information found during the FMEA process. Figure 2.2 shows a standardized empty FMEA worksheet.

Fault Tree Analysis (FTA)

Besides the FMEA, the first stages of the development of safety-critical systems also requires the design of their Fault Tree Analysis (FTA). The FTA is a quantitative reliability analysis developed from a static view of the system. Similarly to the FMEA, the international IEC 61508 safety standard [IEC10] also recommends performing the FTA of the systems at the first stages of the design.

A fault tree is a diagram that displays the interrelationships between a potential critical event in a system and the causes for this event. The approach uses boolean logic to develop comprehensible diagrams, where causes and effects are linked by boolean logic (*and* and *or* gates). The FTA follows a top-down approach. The analysis begins at the system level, where an unwanted failure event is defined as the top event of the fault tree. Then, the subsystem failure events that can lead to that top event have to be identified and defined as sons of the top event. The process continues down the tree until a basic failure (usually a component failure) is found. Figure 2.3 shows an example of a fault tree.

Failure Mode and Effects Analysis Worksheet

Process or Product: ______________________ FMEA Number: __________

FMEA Team: ______________________ FMEA Date: (Original) __________

Team Leader: ______________________ (Revised) __________

	FMEA Process												Action Results				
Line	Component and Function	Potential Failure Mode	Potential Effect(s) of Failure	Severity	Potential Cause(s) of Failure	Occurrence	Current Controls, Prevention	Current Controls, Detection	Detection	RPN	Recommended Action	Responsibility and Target Completion Date	Action Taken	Severity	Occurrence	Detection	RPN
1																	
2																	
3																	
4																	
5																	
3																	
4																	
5																	

Figure 2.2: Standardized blank FMEA worksheet

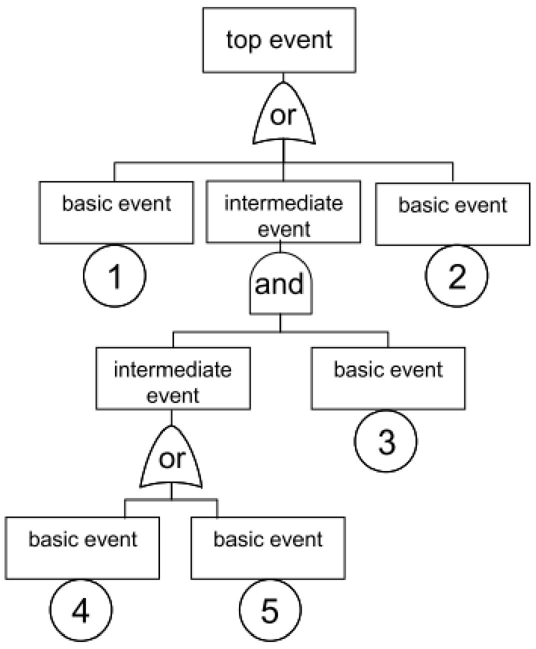


Figure 2.3: A fault tree (from [BKSZN10])

The aim of the FTA is to reveal hidden failures caused by combinations of faults or errors, and quantitatively calculate their occurrence probability. Fault trees can be formally analyzed with mathematical techniques. This way, the probability of the top failure events can be calculated by applying combinatorial techniques to the probabilities of their basic component failures.

Testing

Testing is an analytic means for assessing the quality of systems [UL07], and is one of the most important phases of the development of embedded systems. As Dijkstra pointed out in [Dij70] "testing can be used to show the presence of bugs, but never to show their absence". Therefore, testing is an activity that aims to improve the quality and increase our confidence in a system by identifying defects in them. Testing cannot be undertaken in isolation. Instead, in order to be in any way successful and efficient, it must be embedded in an adequate system development process and have interfaces to the respective sub-processes.

However, although testing is not capable of proving the correctness of a system but only bring out its bugs, it is a widely used technique. Actually, according to [JTM07], "testing is indispensable, and no software system can be regarded as dependable if it has not been extensively tested". This happens, according to Hoare [Hoa96], due to the fact that the contribution of testing is not limited to expose bugs; In fact, its biggest contribution is the feedback that it provides to the development process. In his words, "the real value of tests is not that they detect bugs in the code but that they detect inadequacies in the methods, concentration, and skills of those who design and produce the code." In practice, this means that when a subsystem fails too many tests, the developers do not simply attempt to patch the code. Instead, they look at the development process to determine where the error that eventually resulted in the failure was introduced, and they make the correction there. This might involve clarifying requirements, reworking a design, recoding one or more modules, and in extreme cases, abandoning the entire development and starting a new one from scratch.

A thorough test process is carried out in five different stages [BSI98, IST14, ZN08]:

1. Test planning: Includes the planning of resources and the laying down of a test strategy: defining the test methods and the coverage criteria to be achieved, the test completion criteria, structuring and prioritizing the tests, and selecting the tool support as well as configuration of the test environment.

2. Test specification: Is defining the corresponding test cases using the test methods defined by the test plan.

3. Execution of the test: Carrying out the test cases and test scenarios.

4. Recording the results: Test records serve to make the test execution understandable for people not directly involved (e.g., customer) and prove afterwards whether and how the planned test strategy was in actual fact executed.

5. Checking for completion and test closure: Consists of collecting the test data to consolidate experience, testware, facts, and numbers, and evaluate the test process and generate the necessary documentation.

Validation and Verification

Besides testing, validation and verification (V&V) are especially important in the domain of safety-critical systems due to their high dependability requirements (e.g., safety, reliability, security). Usually, these two terms are defined by two questions:

- Validation: Am I building the right system?
- Verification: Am I building the system right?

In other words, validation is concerned with checking whether the system meets the customer's actual needs, whereas verification is concerned with whether the system is well-engineered, error-free, consistent, etc. That is, verification helps to determine whether the software is of high quality, and validation ensures that the system is useful for the customer.

Fault Hypothesis

A Fault Containment Unit (FCR) is a collection of components that operates correctly regardless of any arbitrary logical or electrical fault outside the region [LH94]. Further, a fault in an FCR cannot propagate and cause a failure outside the FCR. Each FCR in a system must have independent power and clocking resources in order to properly define its boundaries, since shared hardware resources compromise the independence of FCRs.

Given that an embedded system may have more than one FCR, a fault in an FCR that causes a failure in that FCR becomes a fault for the system, due to the causality chain reviewed in section 2.1 (Fig. 2.1). Therefore, dependable systems must tolerate such faults. This fault-tolerance is achieved with the implementation of fault-tolerance mechanisms.

A typical fault-tolerance mechanism for dependable embedded systems is the N-modular redundancy architecture. N-modular redundancy composes the system with N replicated components that perform the same function and provide the same service. A unique service result is obtained by a voting component, which takes a decision by reading the service outcome provided by all the replicated components and executing a predefined voting algorithm (e.g., majority voting). Triple Modular Redundancy (TMR) is a widely used N-modular redundancy type.

Fail-Safe and Fail-Operational Systems

Dependable, and more specifically, safety-critical systems, are typically subdivided into fail-safe and fail-operational systems. Most dependable systems enable to identify one or more safe states, which the systems should be able to reach in case of failure of one of its components. If such a state exists in a dependable system, the system is considered fail-safe. For example, if a failure is detected in a railway signaling system, it should be possible to notify all the drivers of the trains by lighting a red semaphore, which would make them brake and stop their trains, thus leading the system to a safe-state.

Other systems, however, do not have a safe state to which they can safely transit in case of failure. In these systems, at least an application of the computer that is controlling the system must remain operational and keep providing its service, even if it is in a degraded-mode service level. These systems, e.g., flight controls of aircrafts, are called fail-operational.

It must be noted that fail-safeness is not a property of the controlling computer system, but of the controlled system itself.

2.2 Fault Injection

Fault Injection is a widely accepted technique for the validation of the dependability of a system by observing its behavior in presence of faults. The technique consists of performing experiments where artificial faults are deliberately introduced in the system and checking the resulting behavior. The occurrence of faults in the system life-cycle is accelerated by their deliberate introduction; as a consequence, the occurrence of errors and their propagation through the system are also accelerated and the response of the system against such faults can be evaluated.

As Hsueh [HTI97] pointed out, the usage of fault injection techniques enables developers to:

- identify dependability bottlenecks,
- study system behavior in the presence of faults,
- determine the coverage of error detection and recovery mechanisms, and
- evaluate the effectiveness of fault-tolerance mechanisms and performance loss in case of faults.

Faults are typically characterized in three different groups according to their duration [ZAV04]:

- Permanent faults: These types of faults are caused by irreversible damages in the hardware or software, so that they cannot be fixed unless the faulty component is replaced.
- Transient faults: These faults are caused by specific environmental conditions, and they rarely induce permanent damages to the affected components. Therefore, transient faults usually disappear when the conditions change. According to studies in the field, transient faults occur more often than permanent ones, and they are hardest to detect.
- Intermittent faults: These faults are the ones caused by unstable hardware or varying hardware states, and they behave as a sequence of transient faults. They are fixed by the replacement of the faulty component.

Although this characterization of faults in three groups according to their duration is the most common one, other approaches extend this with additional types, such as Kopetz in [Kop06]:

- Soft-permanent faults: The difference between permanent and soft-permanent faults is that permanent faults remain active until the affected HW component is replaced, whereas soft-permanent faults remain active until the state of the HW component is updated.

- Massive transient disturbances: These faults occur when an external event results in the correlated failure of two or more communication channels and possibly some node.

Fault injection techniques are divided into two main groups: execution-based and simulation-based [BP03]. Execution-based techniques consist of executing the system itself (or a prototype) and injecting faults in it during its execution. These techniques are oriented to evaluate the final design, but fixing an eventual design error in them is in general difficult, since the system is already in the latest stages of its development. On the contrary, simulation-based techniques introduce faults in models of systems instead of in the system itself. Simulation-based techniques take more time than execution-based ones, since the formers require to simulate the entire system. However, simulation-based techniques may be applied far earlier in the development process than execution-based methods, what facilitates fixing design errors detected during the fault injection activities.

Fault injection experiments are also typically classified as intrusive and non-intrusive. Intrusive techniques are those in which testing mechanisms leave some kind of footprint in the systems behavior, e.g., an alteration of its time properties. Non-intrusive techniques are the ones that completely mask their presence, so that they have no effect on the system behavior besides the faults they inject. In general, non-intrusive techniques provide more reliable results than intrusive ones, due to the fact that the formers do not affect to the behavior of the system (other than the injected faults). However, intrusive mechanisms are easier to implement than non-intrusive ones. In fact, in some cases non-intrusive techniques may be impossible to implement.

Basic Fault Injection Environment

As Figure 2.4 shows, a representative fault injection environment is composed of 8 main components [HTI97, ZAV04]:

- Controller: Controls the experiment.

- Fault injector: Is responsible for injecting the desired faults into the system under test.

- Fault library: Provides a set of fault types to be injected in the system.

- Workload generator: Generates the workload in order to simulate the desired test case.

- Workload library: Contains a set of workloads to be used by the workload generator.

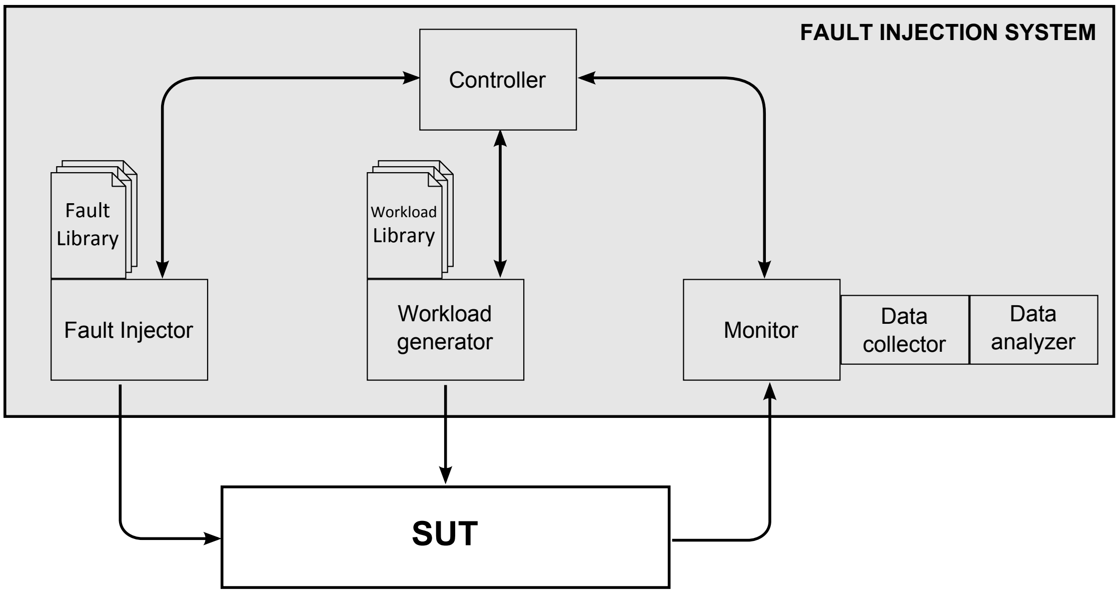


Figure 2.4: Representative Fault Injection Environment

- Monitor: Tracks the execution of simulation commands.
- Data collector: Collects the resulting data during the simulation.
- Data analyzer: Processes and analyzes the data: When the data analyzer works online, the controller might take into account the results provided by the analyzer in order to adapt the fault injection activities and the test cases according to the previously obtained results.

Fault injection techniques are generally divided into three sub-categories, depending on the design phase in which they are implemented: hardware-based fault injection (HWFI), software-based fault injection (SWFI) and simulation-based fault injection (SFI) [BP03].

Hardware Based Fault Injection

The hardware based fault injection (HWFI) involves executing a system with specially designed test hardware that allows injecting faults into the system. HWFI is carried out in the latest stages of the product development, since it needs a functional prototype of the target hardware to be exposed to faults. The hardware prototype is tested by exposing it to physical disturbances, e.g., electromagnetic interferences, heavy ion radiation, pin forcing, which may lead to the destruction of the prototype.

The testing environment for HWFI requires additional hardware facilities to perform the fault injection and to store the resulting data. Hsueh [HTI97] identifies two different types of hardware fault injection depending on the hardware facilities used: Hardware based fault injection with contact and hardware based fault injection without contact.

Hardware based fault injection with contact

The fault injector has direct physical contact with the system under test and produces current or voltage changes in it. Since faults are directly managed by the fault injector, their duration and location is straightforwardly controllable. There are two main techniques to inject faults at pin level:

- Active probes: A flow of current is added to the selected pins via the probes attached to them. This technique is specially suited for stuck-at faults and bridges between two or more pins. This technique may cause damages in the target hardware if two non-compliant pins are bridged or an inappropriate amount of current is added to the pin.
- Socket insertion: This technique inserts a socket between the target hardware and its circuit board. The socket forces the analog signals that represent the desired logic values onto the pins of the target hardware, thus injecting the desired fault. Signals can be inverted, ANDed or ORed with adjacent pins, resulting in stuck-at, stuck-open or even more complex faults.

Hardware based fault injection without contact

In this type of hardware based fault injection, an external source exposes the target system to heavy-ion radiation, electromagnetic interference, anomalous thermal conditions or any other physical phenomenon that could cause internal spurious effects in the target hardware. Therefore, the fault injector has no physical contact with the system under test.

The main benefits of hardware based fault injection are listed in the following:

- Testing experiments provide precise and reliable results, since the methods are non-intrusive and no hardware abstraction is used.
- Test execution is fast (in general, faster than simulated fault injection).
- The system does not need to be modeled.

However, HWFI has several drawbacks:

- It may cause damages to the target platform.
- Accessibility to internal components is very limited.

- Additional hardware is required.
- Observability is very limited.
- Controlability of HWFI without contact is very low, since the precise location and occurrence instant of faults is not controlled by the fault injection tool.
- HWFI without contact is not repeatable, since there is not a precise control in the injection of faults.
- The fixing of bugs that are discovered in the latest stages of the development may be very expensive, since it may require a major re-design of the system.

Software Based Fault Injection

Software based fault injection (SWFI) is a technique that modifies or extends the software implemented in the system under test in order to provide the capability to modify its state while the system is operational. Since faults are directly applied in software, this type of fault injection does not require any additional hardware.

The SWFI technique only enables injecting faults in locations which are accessible by software. It enables introducing both hardware-related and software-related faults to the system, and a wide range of different faults can be applied to it, such as replication or loss of network messages, erroneous flags and error conditions, faults in memories, faults in registers and faults in the operating system. This type of fault injection is typically used in complex systems where interactions are understood through the details of implementation [BP03, PAaP10].

A prototype of the target hardware executes the software that has been modified / extended for SWFI. Since there are some differences between the original code and the modified one, the timing properties and the workload of the system can be affected, and in some cases this may result in behavioral differences when compared to the original software. For example, in case a faulty network channel is simulated by not sending any message through this channel, it may be the case that *‘send’* commands execute in shorter time than they used to take in the original software. This modification of its timing properties may have some collateral influence on the overall behavior of the system. Therefore, this technique is considered intrusive.

According to [HTI97], SWFI is classified in two groups, depending on the moment when faults are injected.

Compile-time injection

This method injects errors by modifying the source code or assembly code before the program is loaded and executed on the target platform. This way, an erroneous software image is loaded into the platform and the effect of hardware or software faults is emulated when the image is executed. This technique is very well suited to emulate permanent faults, but it does not allow to trigger faults during program runtime.

Runtime injection

Additional commands are included in the target software which allow triggering different types of faults during runtime. The most common triggering mechanisms are:

- Time-out: A timer with a predefined time is placed in the code, which generates an interrupt when it expires. This interruption is used as a trigger that invokes a fault. The trigger can be both hardware or software based. The effect of the fault is in principle unpredictable, since the trigger is based on time instead of on the internal state of the program. This method is suitable to inject transient and intermittent faults.
- Exception/trap: The fault is triggered by a hardware or software exception. This method is well suited to trigger faults depending on the occurrence of a certain event, such as the entrance of the program into a certain state (software exception) or when a particular memory location is accessed (hardware exception).
- Code insertion: Instructions are added to the target software allowing fault injection to take place before particular instructions occur. This method is similar to compile-time injection, but faults are injected during runtime.

The main advantages of SWFI are:

- A wide range of faults can be injected, both hardware-related and software-related.
- No additional hardware is required. Implementation cost is low.
- Experiments are fast, almost real-time.
- Experiments are repeatable (except for the case of time-out SWFI).
- The system does not need to be modeled.

However, this technique also has the following disadvantages:

- The software running on the target hardware is not the final software, and this difference may influence the timing behavior of the system.
- Observability is limited, since many times there is no access to internal variables.
- Faults can only be injected in locations that are accessible by software.
- The time needed to implement each experiment may be long, since it may require re-writing and re-compiling code.

Simulated Fault Injection

The simulated fault injection (SFI) technique (also known as simulation-based fault injection) consists of developing a model of the system under test at different abstraction levels, and inserting faults into the model instead of inserting them to the system itself. Since no hardware prototype is needed, this technique can be applied at early stages of the system development, which becomes a major benefit for designers since it reduces the risk of a late discovery of design flaws.

Three different methods are used by the community to inject faults in simulated models [BP03, GBGG01, MVS07]:

- **Simulator commands**: Uses commands provided by the simulator to modify the values of signals and variables at simulation time. The main advantage of this technique is that it does not require modification of the code. However, the usage of this technique depends on the functionalities offered by the simulator. The simulator commands work temporally, i.e., succeeding actions in the model may overwrite the variables. Therefore, fault injection by simulator commands is a good technique for simulating transient faults, but permanent fault injections are not covered [MVS07].

- **Saboteurs**: A saboteur is an additional component that is added to the system, which is capable of modifying the value or timing characteristics of signals or variables. The injection of the fault is managed by a control signal, which determines both the trigger instant and the duration of the fault. The main disadvantage of this technique is that the number of control signals grows as the quantity of saboteurs added to model grows, and this adds additional complexity to the model.

- **Mutants**: A mutant is a component that replaces another component of the system. When the mutant is active it behaves like the component in presence of faults, and when it is not activated it behaves like the original component. An additional control signal is used to trigger mutants. The main disadvantage of mutants is that, like for saboteurs, the number of control signals may be high if several different mutants need to be placed in the model. Moreover, the need to save the internal state of the system before commuting a mutant from non-active to active state may lead to high cost in space ant time.

Simulated fault injection can be applied to models at different levels of abstraction. The accuracy of the simulation and the detail of the faults to inject directly depends on the abstraction level of the model.

The following list summarizes the main advantages of simulated fault injection:

- SFI can be applied to models at different levels of abstraction (more or less detailed models).
- There is no risk to cause damage in the system, since no prototype is used.
- SFI enables an early dependability assessment and therefore reduces the risk of a late and expensive discovery of safety related pitfalls.
- Controllability of fault injection experiments (e.g., location, timing properties) is high.
- Observability is high (depends on the level of abstraction of the models).
- It may be designed as a non-intrusive technique.

On the other hand, SFI presents the following drawbacks:

- The models of the system must be realistic. This sometimes requires an assessment of the models themselves, in order to guarantee a certain level of accuracy between the models and the systems they represent.
- Models at low abstraction levels (e.g., switch-level models) may require long simulation times.
- Models at high abstraction levels (e.g., ESL models) are faster to simulate, but they are not suited to simulate detailed faults.
- Finding a balance between an acceptable simulation time and an adequate abstraction level may not be straightforward.

2.3 Cognitive Complexity

IEEE defines the term complexity as "The degree to which a system or component has a design or implementation that is difficult to understand and verify" [IEE91]. Due to the increasing number of services that safety critical embedded systems provide, these systems are exhibiting a considerable degree of complexity. For example, high-end cars had about 70 electronic control units (ECUs) back in 2006 [Bro06], and this number raised up to 100 for the year 2011 [BCR$^+$11] (see Figure 2.5). In order to deal with this complexity challenge and make the systems as comprehensible as possible, the researching community has identified three main strategies of simplification that can be applied to embedded systems [Joz01, Kop08, Per11]:

- **Abstraction**: Construction of a higher-level concept that captures the essence of the problem-at-hand without taking into account irrelevant detail.

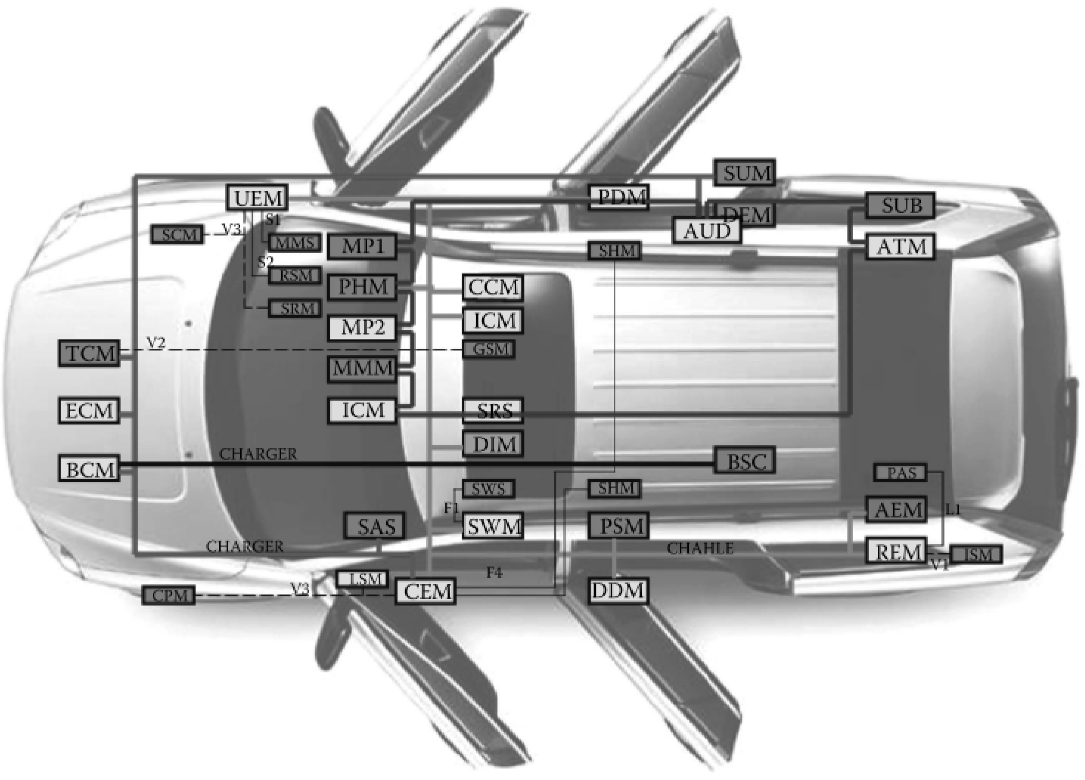


Figure 2.5: Network infrastructure of the Volvo XC90 (from [HNSS09])

- **Partitioning**: Decomposition of the problem scenario into (nearly) independent parts, which that can be studied individually. It is also know as *separation of concerns*.
- **Segmentation**: Division of the complex behavior of the system into smaller parts which are simpler to understand, and that can be processed sequentially. It is also called *temporal decomposition*.

It is worth to note that in some cases it is impossible to apply some of the simplification strategies mentioned above. For example, segmentation may be impossible if the behavior of the process is formed by highly concurrent processes, or if it depends on feedback loops that cause complex dependencies between variables.

Integrated architectures are a good example of the three strategies stated above. They may include a chip, which is in fact a powerful abstraction of more than a billion transistors working in parallel and communicating with each other. Following a partitioning strategy, integrated architectures are typically treated as a set of separate components. Furthermore, the explicit notion of time of each integrated architecture enables the segmentation into sequential tasks [AaB12].

The term cognitive complexity describes the cognitive resources required to perform a task [Rum06]. The research made by Cowan in [Cow01] identified that in the case of human beings, our memory capacity limits our comprehension capabilities to the processing of four simultaneous chunks of information. Therefore, although functionality of a system may be *simple* from a complexity point of view, designers must ensure that the techniques employed to achieve the means of dependability, such as the introduction of fault-tolerance mechanisms, do not affect the simplicity of the system significantly. Redundancy of components enables the separation between the normal functionality of the system, and it is therefore straightforward to remove while the functional properties of the system remain intact. Therefore, redundancy of components should be preferred against other techniques that require mixing functional aspects of the systems with non-functional ones (e.g., state-recovery strategies).

2.4 Model Based Design (MBD)

In order to handle the increasing complexity of safety-critical embedded systems, their design is nowadays usually performed by applying the so-called Model Based Design (MBD). According to the Oxford English Dictionary [Oxf07], a model is "a simplified description, especially a mathematical one, of a system or process, to assist calculations and predictions". Therefore, the essence of modeling lies on focusing on a (or some) specific property of a component in order to reason about it, whereas being agnostic of the properties that are not interesting for the actual purpose. This simplification increases the understandability of the system.

MBD [SK97, SK01, GKL+10] is a system design approach that advocates the notion of using high level models throughout the development of a system. Model-based engineering (MDE) reinforces this notion by promoting models not only as a mechanism for abstraction, but also for verification, implementation, testing, and maintenance.

MBD raises the level of abstraction in system design from low-level languages to high-level modeling formalisms. In many cases, high-level models treat concepts like concurrency and time as first-class notions, which enables to design systems that would be hard or impossible to design using low-level methods. In these cases in which timing properties of models are defined non-ambiguously, automatic tools might be used to generate corresponding software code directly from the models. Commercial examples of such modeling and code-generation frameworks include Simulink Coder [wwwy] from The MathWorks, which generates code from Simulink models, LabVIEW C Generator [wwwr] from National Instruments, and SCADE Suite [wwww] from ANSYS-Esterel Technologies. Academic tools include Ptolemy II [wwwt] from the University of Berkeley and the Distributed Operation Layer (DOL) [wwwg] from ETH Zürich.

Meta-Model

Several different definitions have been proposed for the concept of a meta-model. In our opinion, the most accurate ones have been provided by J. Estublier and I. Kurtev [EBF05, Kur05]:

- "A meta-model is the model of a set of models" [EBF05]
- "A meta-model is a model of a modeling language" [Kur05]

All in all, a meta-model is the set of rules and constructs under which a model can be built, so that a model is usually considered a specific instance of the meta-model.

The Model Driven Archicteture (MDA)

The Model Driven Architecture (MDA) [MM03] is the standard proposed by the Object Management Group (OMG) for the model-based development of systems. The main aim of the MDA is to increase the portability, interoperability and reusability of the models. To achieve that goal, the MDA suggests separating the specification of the operation of a system from the details of its deployment on a specific platform. Therefore, the MDA decouples the system in two models: the Platform Independent Model (PIM) and the Platform Specific Model (PSM). Different modeling languages, such as the Unified Modeling Language (UML) [OMG11a] the Systems Modeling Language (SysML) [OMG12], and the UML profile MARTE [OMG11b] are compliant with the MDA. These modeling languages are further discussed in Section 3.1.

The Y-chart Paradigm

The Y-chart paradigm [BCG$^+$97, KDVvdW97, KDvdWV02, LTS$^+$09] is another model-based design approach for embedded systems, which has been very extensively used lately, although it has been implemented in many different ways [LTS$^+$09].

The Y-chart approach is a methodology to provide quantitative data about the performance of architectures for a given set of applications. This work must not be confused with the Y-Chart methodology by Gajski and Kuhn [GK83, GAGS09], which is a different research work that provides a taxonomy for visualizing the design of the systems from three different hierarchical points of view.

Similarly to the MDA, the Y-chart modeling approach relies on separating the functional and platform-related aspects of the system in order to deal with the increasing complexity of systems. As Figure 2.6 shows, the development of systems with the Y-chart approach starts from a separate definition of the application and the platform. Once both models are finished, the components of the application are mapped to the platform and the system is analyzed by simulation. The analysis of the results is used to improve or fix different aspects of the application, the platform, or even the mapping.

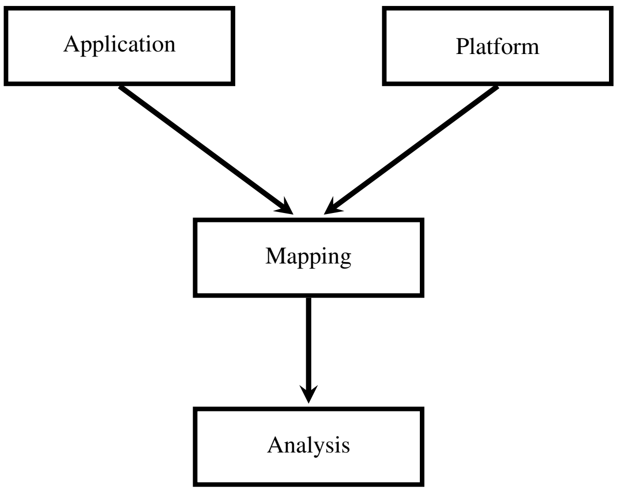


Figure 2.6: Design work-flow of the Y-chart paradigm.

One of the biggest benefits of this approach is that a number of different mappings can be defined and simulated for a given application and platform, which enables designers to compare and evaluate diverse versions of their systems with regard to different aspects (e.g., the suitability of a platform for a given functionality, the efficiency of each system candidate) in order to choose the most appropriate system for their purpose.

Although the key concept underlying the Y-chart paradigm, constisting of explicitly separating application descriptions from a platform specification, is relatively simple, different methodologies have implemented it in different ways. The most significant examples are POLIS [BCG+97] and its derivative Metropolis [BWH+03], Intel's CoFluent Studio [wwwd], or the Distributed Operation Layer [wwwg] and its continuation, the Distributed Application Layer (DAL), which are further discussed in section 3.3.

2.5 The Notion of Time

The notion of time is a key concept in embedded systems design [Per11]. This chapter reviews the notion of time from different perspectives, identifying the most extended time representations and models of computation (MoCs).

Time Flow

In the field of embedded systems, the concept of time is usually based on its Newtonian notion, so that relativistic effects are omitted. In this case, the flow of time can be modeled as a straight timeline that connects the past with the future, as figure 2.7 shows. An *instant* is a cut that divides the timeline in two parts, whereas the *present* is a special instant that separates the past from the future. The *duration* is the time lapse between two instants, and an *event* is a relevant happening that takes place at an instant [OESHK07].

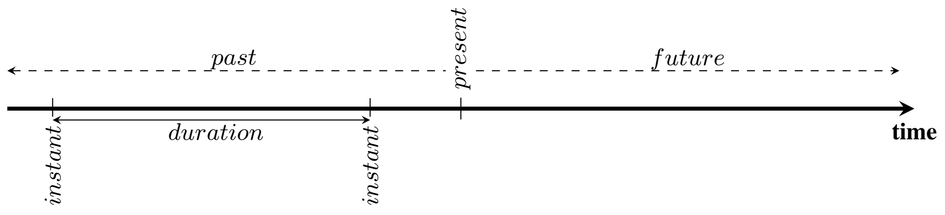


Figure 2.7: Time flow

Models of Time

Depending on the field of application, time may be abstracted in different models in order to reduce its cognitive complexity [Per11]. This section reviews the most widely used models of time.

- *Continuous / Dense-Time model*: The concept of time in everyday life is usually based on the continuous/dense-time model (Figure 2.8a). The dense model of time is based on a continuous flow, where time advances continuously and the delay between two events can be arbitrarily small. Analog electronic systems are based on this model.

- *Discrete-Time model*: The discrete-time model (Figure 2.8b) is an abstraction of the dense time model, where time is considered to advance by discrete steps, instead of continuously. Time instants are modeled as positive integers. Events only occur at discrete-time values, and the duration between two events is a multiple to the time step. The discrete time ticks are usually generated by a clock that generates periodic events called microticks. This model of time is used by synchronous languages (e.g., Lustre, Esterel) and hardware description languages (e.g., VHDL, Verilog).

- *Global-Time model*: The global-time model (Figure 2.8c) is widely used in distributed real-time embedded systems due to its capability to provide a consistent temporal order of events based on their time-stamps, which eases to establish a consistent execution of control algorithms partitioned in distributed systems. The global-time is approximated by generating a macrotick using the local microtick clock of the distributed computers and a clock synchronization algorithm [Kop11]. This approach achieves a maximal divergence of one macrotick among the local clocks, which is considered the reasonableness condition.

- *Sparse-Time model*: The sparse model of time (Figure 2.8d) takes the concept of global-time and establishes alternating intervals of activity (π) and silence (Δ) over time in order to provide a wide notion of simultaneity to distributed systems. The occurrence of events that are under the sphere of control of computer systems is restricted to happen during the activity intervals. All the events that occur within the same activity interval are considered simultaneous. This time abstraction suits distributed real-time embedded systems that require to guarantee a deterministic temporal order of events and a consistent notion of simultaneity overall the system.

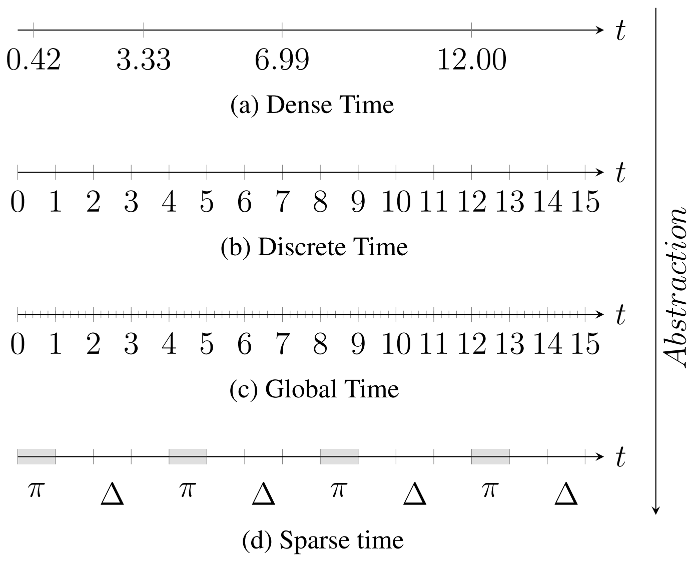


Figure 2.8: Models of time (from [Per11])

Models of Computation

Although the concept of 'Model of Computation (MoC)' has been defined in many different ways by different authors [ELLSV97, LSVS98, LSV98, LN05, Jan09, Mar11], it is generally accepted that a MoC is an abstraction of a real computing device that gives a description of the mechanism assumed for performing computations, i.e., it gives semantics to the structure models. A MoC abstracts slightly from the languages and allows the designers to focus on the essential issues of concurrency, time, communication and synchronization [JS05]. Therefore, MoCs always omit some properties and details that are irrelevant for their purpose and focus on other properties that are essential.

However, as Lavagno et al. point out in [LSVS98], the distinction between a language and its underlying model of computation is important. The MoC affects the expressiveness of a language, whereas the syntax affects compactness, modularity, and reusability. Thus, the same MoC may be shared by different languages, and some languages may support more than one MoC.

According to Jantsch [Jan09] , models of computation for embedded systems should not address principle questions of computability or feasibility but should rather aid the design and validation of concrete systems. Similarly, Henzinger [Hen04] states that the MoC has a crucial importance when simplifying the design and implementation of embedded software. Lee [Lee00] points out that the composition of elements must have consistent, non-ambiguous and non-conflicting temporal properties, especially in the case of fault-tolerant systems.

All in all, it is clear that the choice of the most suitable model of computation is a key decision in the design of embedded systems, since different MoCs may lead to significant differences in terms of complexity and in the effort needed for verification and validation. This choice of an adequate MoC takes an even greater importance in the case of safety-critical systems, since the reduction of the cognitive complexity and the capability of performing verification and validation activities play a crucial role in the certification of systems.

The models of computation have been evolving during the history of computing. Brooks et al. [BLL$^+$08], Jantsch [Jan09] and Lee [Lee02] provide a good overview of the different models of computation used nowadays. The following sections briefly introduce the currently most relevant models of computation, ordered by the timing abstraction they use.

The Continuous-Time MoC

The continuous-time MoC has been most widely used in the modeling of mechanical and fluidical dynamics, analog circuits, chemical processes and many other physical systems. This type of continuous dynamics are typically defined as ordinary differential equations (ODEs), expressed in some mathematical notation. The idea of this MoC is that, as physical entities, the signals in the continuous MoC have continuously evolving values that can be observed at any instant of the dense time.

Obviously, since digital computers rely on discrete timestamps provided by logical clocks, continuous-time models must be approximated by solvers, such as the Euler integration method or the Runge-Kutta algorithms [But03,CK06]. By using these types of solvers, continuous systems can be simulated similarly to discrete-event models, with the only difference that, in addition to using an event queue to determine the advance of time, an ODE solution has to be computed at each time step.

Design and simulation tools such as Spice [Nag75], SystemC-AMS [IEE13], Simulink [wwwx] and Dymola [wwwh] are based on the continuous MoC.

The Discrete-Event (DE) MoC

Models based on the Discrete-Event (DE) MoC [Fis01] rely on a global notion of time along the system, and the occurrence of some time-stamped events during the execution. Systems based in the DE MoC behave reactively, i.e., they perform some computation as a reaction to some event. Thus, data in DE systems can only change with the occurrence of events, and not continuously over time.

Hence, the discrete-event MoC is well suited to model systems where the information or state changes by the eventual occurrence of certain discrete phenomena. Examples of this kind of systems are manufacturing plants, where actions have to be performed when a component reaches the end of the conveyor belt; automated inventory systems, where cranes and robotic arms move until a presence sensor detects their location; or communication systems, where the reception of some information package is considered the event to start performing some computation or answering the message.

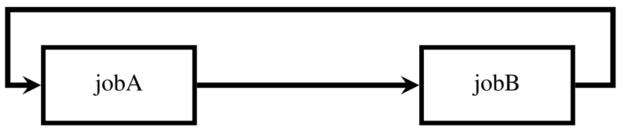
Some discrete-event based models restrict the occurrence of events to time-stamp events that take place every predefined time interval. This variant of the DE MoC is usually known as the Discrete-Time (DT) MoC. Hardware description languages such as Verilog [TM02, IEE06] and VHDL [IEE09] are based on the DT MoC.

The Synchronous Reactive (SR) MoC

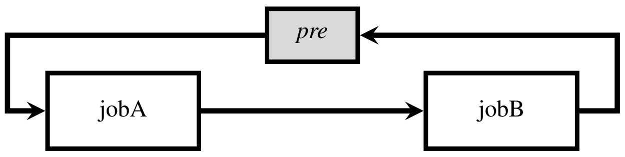
The idea of synchrony in software was first introduced during the early eighties in some theoretical works by Milner [Mil83]. However, the synchronous approach [BB91] was born in the beginning of the nineties by the French school with the development of the Lustre [HCRP91], Esterel [BG92] and Signal [LGGLBLM91] synchronous languages.

Synchronous languages adopt the synchronous hardware design paradigm in the sense that they assume that computation occurs in two separate phases, computation and communication, that do not overlap. This is ensured by the fact that the critical path of the combinatorial hardware between latches takes less time than the clock cycle. Therefore, the behavior of the hardware is independent of the delays of the combinatorial logic.

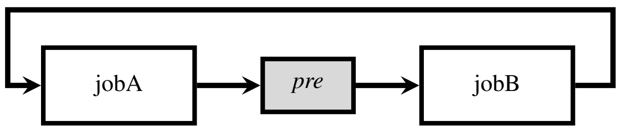
The synchronous reactive MoC relies on the synchrony hypothesis, which analogously to synchronous hardware, enables abstracting the temporal concerns from the functionality of the system. This is achieved by considering the systems to be ideally reactive, that is, they react to input stimuli producing their outputs in no observable time. Therefore, it is usually said that both computation and communication take zero time in the SR MoC. However, a more accurate definition would be to say that the time taken for communication and computation does not matter from a functional point of view as they do not overlap, since if computation is fast enough, there is no interference with timing concerns.



(a) Loop without *pre* operator. Cannot derive an schedule



(b) Loop with *pre* operator. Schedule: JobA, JobB



(c) Loop with *pre* operator. Schedule: JobB, JobA

Figure 2.9: Partially ordered schedule in SR models

Synchronous reactive models rely on a discrete-time basis, where both jobs and communications are executed instantaneously (in zero simulation time). For their execution, all the jobs read their inputs, execute their functionality and write their outputs instantaneously. Thus, all the inputs of a job must be available when the job is triggered. Since all the jobs are triggered synchronously and communications are also instantaneous, the compiler needs to derive a partially ordered schedule from the system before it is executed, in order to ensure the availability of such inputs during the definition. Hence, defining program loops as the one shown in Figure 2.9a is strictly forbidden in synchronous languages, since it makes it impossible to extract a coherent partially ordered schedule.

In order to deal with this drawback, synchronous languages provide mechanisms to delay the communications between two jobs, such as the *pre* operator of Lustre. By delaying the communications by one cycle, the *pre* operator establishes the starting and finishing points of the cycle of simulation, which enables the scheduler to provide a partially ordered schedule for the model, as shown in Figures 2.9b and 2.9c.

Another drawback of SR models is the difficulty of generating distributable code from them. Embedded applications are frequently distributed over a cluster composed by several computers communicating by means of a communication infrastructure. Since these architectures frequently do not comply with the synchronous execution model, generating a distributable code with the guarantee that it will behave exactly as in the synchronous model is a very challenging task. Different techniques have tried to overcome this problem; however, until now, the most successful methodology is the "Globally asynchronous, locally synchronous system" (GALS) approach [Cha85, BWH+03, HM06, KGGV07], which relies on designing distributed asynchronous systems composed by several synchronous subsystems, with the aim to take advantage of the benefits of synchronous components, whereas trying not to increase the cognitive complexity of the system by attempting to make it fully synchronous.

All in all, the SR MoC enables the designers to abstract their design from temporal concerns. This provides solid mathematical definitions with strong properties, which reduces the state space of expressible designs and eases the formal analysis and synthesis of the systems. This facilitates certification because it reduces ambiguity and makes it possible to construct formal proofs about the operation of the system. Due to this fact, synchronous models are nowadays being extensively used in safety-critical systems, with commercial tools such as SCADE Suite [wwww, wwwv].

Finally, as pointed out by Benveniste [BB91], it is important to note that synchronous languages are not completely bound to non-determinism. Some of the synchronous languages accept non-deterministic programs as modules, although they refuse to produce deterministic code out of them. This might be very useful to model the environment of the synchronous system, or any other physical phenomenon.

The Time-Triggered (TT) MoC

The Time-Triggered (TT) MoC [EBK03] heavily relies on the sparse notion of time, which was first introduced by H. Kopetz in [Kop92]. As mentioned previously, the main idea behind the sparse-time model consists on establishing an alternating sequence of intervals of *silence* and *activity* along the progression of time, as shown by Figure 2.8d.

When systems are designed in a distributed fashion, establishing a precise synchronization between local clocks is essential in order to maintain a temporal coherence between the distributed components. The quality of this synchronization is measured by two properties:

- *Precision*: The maximum offset between any two clocks in the system during a given interval.
- *Accuracy*: The maximum offset between a given clock and the absolute reference time.

Although a detailed analysis of the properties and technologies of a system allows us to identify the precision of the synchrony between clocks, the continuity of the dense time and the digitalization error make it physically impossible to guarantee that two observations of the same event will always yield the same timestamp in a distributed system; i.e., whatever the precision of the synchrony between two clocks is, it may always happen that a given event is observed by two different timestamps in each clock.

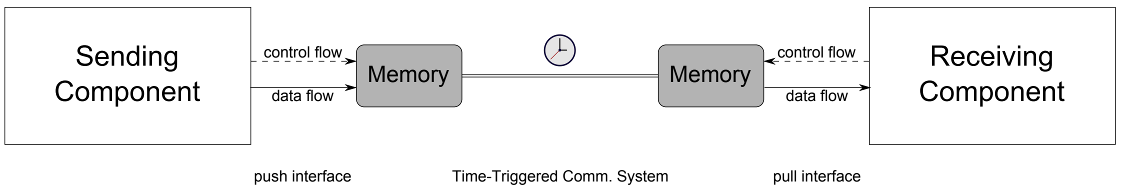


Figure 2.10: Temporal firewall

The definition of such a time-triggered MoC came from the need to overcome this problem.

Models relying on the time-triggered MoC restrict the occurrence of significant events to the activity interval of the sparse-time model. Events occurring at the same timestamp are considered instantaneous, whereas events that are separated by at least one segment of silence are assigned to different timestamps by all the clocks in the system. As a result, the TT MoC makes it possible to establish a consistent temporal order of events all over the system.

Obviously, the occurrence of external events might take place during the silence interval, since they are not under the sphere of control of the system. Therefore, environmental events must be assigned to an activity interval by an agreement protocol in order to maintain the consistency of the system.

Communication between components in the TT MoC is performed by using a so-called temporal-firewall [KN97]. A temporal firewall is an interface for the unidirectional transmission of data from a sender to a receiver. As Figure 2.10 shows, each component of a TT MoC-based system contains a dedicated memory object that acts as its information container. The temporal firewall, which is composed by a time-triggered communication system and the dedicated memory of each component, provides both *push* and *pull* interfaces for communication. This way, components can submit information by writing data into their memory via the *push* interface, whereas components that want to receive some information can read their memory via the *pull* interface. The exchange of data between memories is managed by the time-triggered communication system, and only depends on the progression of time.

In order to guarantee the consistency between the models and the physical systems, the latter must comply with the following conditions:

- The local clocks of the system must be synchronized according to an a-priori known precision, which establishes the minimum duration of the silence interval of the sparse time.

- The architecture must ensure that all the internal events are restricted to the activity interval, and must provide agreement protocols for the ordering of external events.

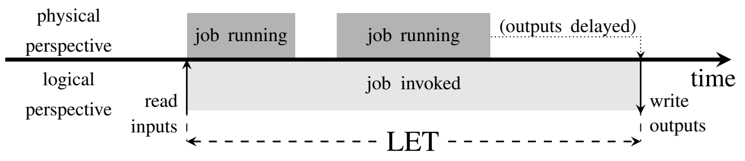


Figure 2.11: LET: Logical Execution Time of a job

- The time-triggered communication schedule must be known by all the components of the system and the communication channel. The communication channel must guarantee that the information sent by a sender reaches the receiver within the a-priori known delivery interval.

When the system is built as a Time-Triggered Architecture (TTA) based system (see section 2.6) and the communication is performed by time-triggered protocols such as TTP [KG93, Aer11c] or TTEthernet [KAGS05, Aer11b], the conditions above are met by construction. In fact, the time-triggered MoC was designed in conjunction with the time-triggered architecture. This makes the time-triggered paradigm a good candidate for the design of dependable hard real-time systems.

The Logical Execution Time (LET) MoC

The concept of Logical Execution Time (LET) [KS12] was first introduced with the Giotto time-triggered programming language [HHK03, HHK01b, HHK01a]. LET is a time-triggered MoC, in the sense that it is governed by the passage of time. However, the main particularity of the LET MoC is that it specifies a logical duration for each computational job regardless of its physical duration. Therefore, the LET paradigm is well suited for time-triggered systems that exhibit time-periodic behavior.

The logical execution time of a job specifies the duration between the instant when the inputs are read and the instant when the outputs are written. Thus, the logical execution time and the physical execution time of a job are different concepts, since the physical execution time focuses on the amount of time needed for computation and the logical execution time focuses on amount of time between the reading of inputs and writing of outputs.

Communication between jobs is instantaneous in LET models. Hence, if the program completes its execution before the deadline of the LET, the update of the outputs is delayed until that deadline. In other words, from the physical point of view, the LET of a job is the upper bound of its execution time. However, from the logical perspective, the LET of a job is not only the upper bound, but also the lower bound. Figure 2.11 shows the LET of a job that is physically executed in two time slots.

Due to the decoupling of physical and logical execution times, the use of faster machines does not result in a logically faster program, but only in decreased machine utilization. In other words, neither the value nor the time-behavior of the LET model get influenced by the speed of the target platform. Thus, when the deployment of the LET model into a hardware platform satisfies the LET specification, the behavior of the system does not vary depending on the platform, and determinism holds in the system. The LET specification is satisfied if the following conditions hold:

- The (physical) worst-case execution time (WCET) of a job is not longer than its LET.
- It is guaranteed that the inputs signals of a job are read at the beginning of its LET interval.
- It is guaranteed that the outputs signals of a job are written at the end of its LET interval.

While the first condition depends to a great extent on the technology of the HW components, the functionality of the job and the programming style, the second and third conditions are inherent when the LET model is mapped onto TTA platforms.

Since its introduction with Giotto, the LET paradigm has evolved and several different modeling languages have been derived from it. The Timing Definition Language (TDL) [Chr14, RDPN10], is a LET based programming language for dependable hard real-time systems. The TDL extends the LET MoC with the capability to include asynchronous activities and the possibility to create platform specific models by deploying the functionality into platform models. The TDL environment is nowadays integrated in the Chrona Creation Suite [wwwc], which is a commercial extension for the MATLAB/Simulink tool for the modeling, simulation and distribution of TDL models.

The Hierarchical Timing Language (HTL) [GSVK+06, KLM08, HKMS09] extends the LET paradigm by enabling the definition of hierarchical LET models. Its main benefit is that schedulability in HTL models only needs to be checked for the highest abstraction level, since each task refinement is constrained in such a way that if the task is schedulable, the more detailed model of the task is also schedulable. Therefore, a time-safe HTL program may be changed locally without the need for re-checking time safety globally.

All in all, the LET paradigm has been widely analyzed and extended since its introduction with the Giotto language. Several simulation environments have been created or adapted to it, such as the native Embedded Machine [HK02], Ptolemy II [BLL+07a, BLL+07b, BLL+08] and Simulink [HKSP03] for Giotto, and the the Virtual Creation Suite / Simulink [wwwc, DNP+10] for TDL models.

The Kahn Process Networks (KPN) MoC

The Kahn Process Network (KPN) MoC was introduced by G. Kahn in 1974 [Kah74]. The KPN MoC is an abstraction that assumes systems are a network of concurrent processes that communicate among themselves via unbounded FIFO channels. Writing semantics are non-blocking, i.e., processes always succeed to write immediately. On the other hand, the reading activity is blocking, i.e., in the case a process wants to read from an empty channel, it would halt and would not be able to continue until the buffer had sufficient tokens to satisfy the read. Each reading activity implies a consumption of the token that has been read.

Each signal has its own queue of ordered events, which are stored in the FIFO channels. However, there is no order relation between events in different signals. Thus, KPN models are not totally ordered, but only partially ordered. For this reason, the KPN MoC is considered an untimed MoC.

One of the particularities of KPN models is that they are monotonic, which means that they do not need all the input information (input stream) to be available to produce an output, but they are capable of producing a partial output stream from a partial input stream.

Monotonicity enables parallelism, since a process does not need all the inputs to start its computation and provide outputs. For this reason, the KPN model has become a very popular MoC to model parallel applications executing on SoCs [HHBT09], due to its suitability to represent both parallel applications and hardware platforms.

Lee et al. [LP02] provide an extensive overview of the KPN MoC and one of their derivatives, the data-flow process networks.

Heterogeneous Systems

Cyber-physical systems (CPSs) [Lee08, LS11] are embedded systems and networks of systems that interact and control physical processes, typically with feedback loops. Thus, the design of such systems requires a good understanding of the joint dynamics of several areas, including computing, networks and different physical processes. This study of the joint dynamics is the key point that distinguishes CPSs from other disciplines.

As mentioned before, choosing the most appropriate MoC when modeling a certain system can dramatically reduce the cognitive complexity of the model. However, in the case of CPSs, where very different computing and physical phenomena interact, it is not realistic to think about a single model of computation that will enable to model the entire system with cognitively simple models, due to the different nature of each sub-system.

In order to deal with this issue, these type of systems, known as heterogeneous systems, are usually divided in different subsystems, where each subsystem is modeled by a different design team formed by experts in the field. Therefore, each of the subsystems typically relies on a different MoC, which complicates the simulation of the complete system.

A number of papers have been published on this topic, e.g. [ELLSV97, LSV98, EJL+03, SJ04, MPSJ06, MPSJ08b]. The following subsections describe the most successful approaches for heterogeneous system modeling and simulation.

Ptolemy II

Ptolemy II [EJL+03, Pto14, wwwt] is an open source graphical modeling and simulation framework for heterogeneous embedded systems. The project is developed by the University of California at Berkeley. Ptolemy II provides a seamless integration between functional parts of the system relying on different MoCs. This integration is accomplished by the notion of *domain polymorphism*.

In Ptolemy II, the term *domain* is used to refer to an implementation of a MoC. The notion of domain polymorphism is introduced by the fact that most of the components (also called *actors* in Ptolemy II) can be executed under the direction of any of the domains defined in the domain-library provided by Ptolemy II. Hence, models of Ptolemy II are considered *domain polymorphic*, since they have a well-defined behavior in more than one domain, and that the behavior is not necessarily the same in different domains [Lee02].

Models are built in a graphical way in Ptolemy II, by composing actors, connecting them and assigning a domain to each of the subsystems. The domain manages the interaction between components and the flow of control, i.e., it provides the execution semantics to the assembly of components. The assignment of a domain is performed by the definition of the so-called *director* components to the subsystems. The key to hierarchically composing multiple models of computation is that an aggregation of components under the control of a domain should itself define a domain polymorphic component. This is why each subsystem must have its own *director* defined.

Among others, nowadays Ptolemy II is able to simulate and provide directors for the following models of computation [HLL+03, Pto14]:

- Continuous MoC: *CT Domain*
- Discrete Event MoC: *DE Domain*
- Synchronous Reactive MoC: *SR Domain*
- LET MoC (Giotto): *Giotto Domain*
- Process Networks MoC: *PN Domain*
- Synchronous Dataflow MoC: *SDF Domain*

- Finite State Machines MoC: *FSM Domain*

[BLL$^+$08] provides an extensive description of each of the domains provided by Ptolemy II.

SML-Sys

SML-Sys [MPS04, MPSJ08b] is a framework for the design of heterogeneous systems that was developed by the researchers of the KTH Royal institute of Technology. SML-Sys is based on the Formal System Design (ForSyDe) [SJ04], which defines a methodology for the design of system models at an abstract high level and their refinement into more detailed implementation models for synthesis. Systems rely on the synchronous reactive MoC in ForSyDe. With the aim of easing the development of heterogeneous systems, SML-Sys extends the approach proposed by ForSyDe by enabling the developers to use different MoCs. Heterogeneity is addressed in SML-Sys by using domain interfaces that add or remove events from event sequences. The SML-Sys is implemented in the Standard ML [Mil97] language.

In contrast to ForSyDe, SML-Sys uses the KPN MoC for the definition of the system at the highest level. When the system is refined into more low-level models, SML-Sys supports more models of computation besides the KPN, such as the clocked, timed and synchronous MoCs. An extension to SML-Sys called EWD [MPSJ08a] even supports some code-generation facilities.

Extensions to SystemC

Several different approaches have enriched SystemC (section 2.7) with the capability to model and simulate systems relying on different MoCs, such as:

- SystemC-AMS [IEE13]: Extension to SystemC for the modeling of systems with analog and mixed (digital-analog) signals (see section 2.7).

- SystemC-H [PS06]: Extension to SystemC for the modeling of heterogeneous models (see section 2.7).

- HetSC [HSV05]: Another extension to SystemC for the development of heterogeneous models (see section 2.7).

- HetMoC [ZSJ10]: Novel extension to SystemC for the design of heterogeneous embedded computing systems (see section 2.7).

- E-TTM [PNOES10a] (see section 3.5): Extension to SystemC for the design of systems based on the Time-Triggered Architecture.

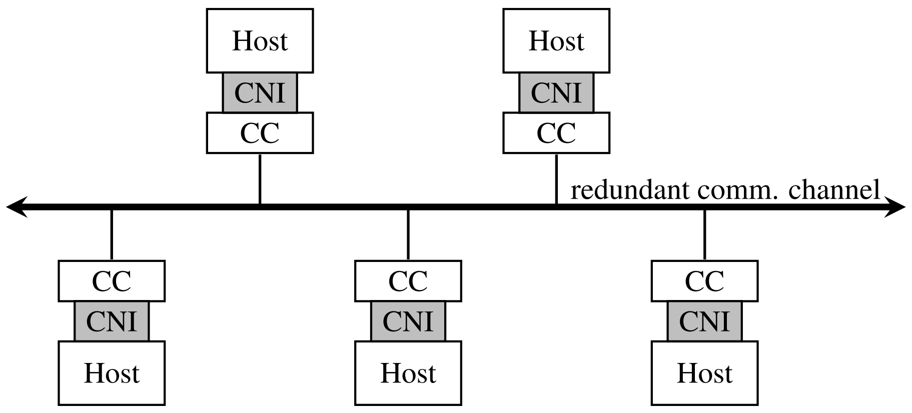


Figure 2.12: Structure of a TTA cluster with five nodes

- SystemC-MDVP (SystemC Multi Domain Virtual Prototypes) [FWI$^+$14]: Extension to SystemC-AMS consisting on an open framework for the modeling of multiple physical domains, such as micro-fluidic systems.

2.6 The Time-Triggered Architecture (TTA)

The Time-Triggered Architecture (TTA) [Kop98a, KB03, Kop11] provides a computing infrastructure for the design and implementation of dependable and safety-critical embedded systems. The TTA decomposes systems into nearly autonomous clusters and nodes that share a fault-tolerant global time base of known precision. The existence of this global time in all the components of the system enables to abstract the communication interfaces, guarantees the timeliness of real-time applications, and eases prompt error detection in communications. Therefore, the TTA is based on the time-triggered MoC [Kop98b], which relies on the sparse-time model of time. The TTA infrastructure guarantees the agreement between the time stamps at each node.

The interfaces and the predictable Time-Triggered Protocol (TTP) decouple the processing functions from communications among the distributed subsystems, thus simplifying the design of the internal application software of the nodes. In the TTA, systems are composed of one or several clusters, which are composed of one or several nodes interconnected by a replicated time-triggered network (Figure 2.12). Each node consists of a time-triggered Communication Controller (CC), a Communication Network Interface (CNI) and a host processor with memory that executes the operating system and the application software.

The communication system (composed by the communication network and controllers) executes periodically following an a-priori specified schedule, i.e., it reads a message from the CNI at the sending node at an a-priori known instant, and delivers it to the rest of the nodes at an a-priori known instant.

The dynamics of the real-time application are modeled by a set of relevant state variables, called real-time entities. RT entities have some static attributes that do not change during their lifetime, such as their name, type or unit, and a set of dynamic attributes, such as their value at a given instant. The observation of an RT entity represents the information about its state at a particular instant, and can be captured in the following data structure:

$$Observation = \langle name, value, t_{obs} \rangle$$

In order to manage complexity, three different types of interfaces were defined in the very first specifications of the TTA [KS03, KB03]. Later versions specify four different interfaces [KOESH07, Kop11] (Figure 2.13):

- **Linking Interface (LIF)**: The LIF is the interface that provides the timely information to the nodes during the operation of the system. This interface is used by the nodes to communicate among themselves, and it is therefore a time-critical interface that must meet the temporal specification of the application in all possible scenarios. This interface is also called *Real-time Service (RS) interface* in [KB03].

- **Configuration and Planning Interface (CP)**: The CP is the interface used to configure the system, i.e., to connect a node to other nodes. It is used during the integration phase to generate the "glue" between the quasi-autonomous nodes. Hence, this interface is not time-critical. This interface is also called *Technology Independent Interface (TII)* in [Kop11].

- **Diagnostic and Maintenance Interface (DM)**: The DM is the interface that enables maintenance engineers to observe the internal state of the nodes and set their internal parameters. This interface does not influence the temporal behavior of the nodes, and it is usually not time-critical. This interface is also called *Technology Dependent Interface (TDI)* in [Kop11].

- **Local Interface**: The local interface connects the component to the external world. This interface is used to link the system with the environment.

The aim of the TTA is to design large real-time systems. Despite the fact that a large system will support many more functions than a small system, the complexity of each individual function must not increase with the growth of the system [KB03]. To that end, the only central element of the TTA is the global notion of time. As seen in section 2.3 one of the ways to handle the cognitive complexity of systems is to enable the designers to divide the system into different subsystems (partitioning). The TTA supports the definition of subsystems by the inclusion of gateway nodes in the system. Gateway nodes are special nodes that contain two CNIs, and they are used to connect

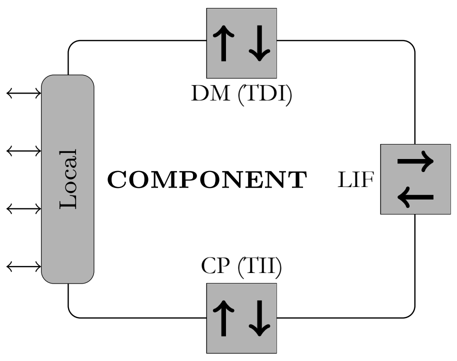


Figure 2.13: Four interfaces of a TTA component

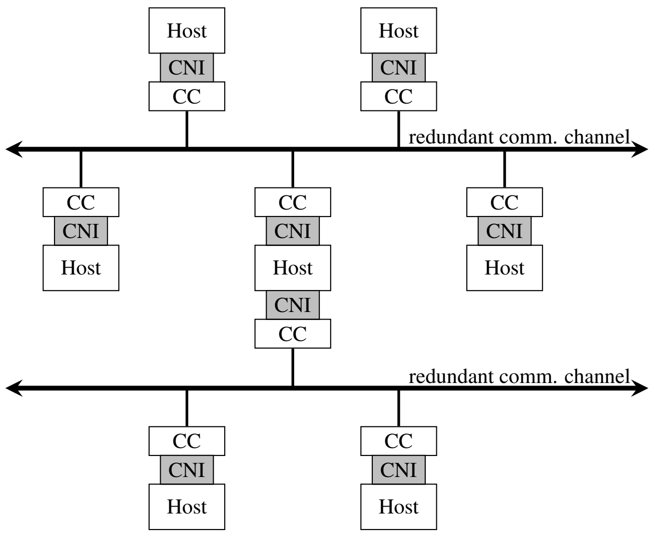


Figure 2.14: Structure of a TTA system with a gateway

different subsystems. The example in Figure 2.14 shows how a node can be expanded into a gateway node when its computational limits are reached. This way, the interface to the original cluster remains unchanged in the value and time domains, whereas the functionality of the node is now distributed in a second cluster. Gateway nodes are also useful to integrate legacy systems.

2.7 SystemC

SystemC [IEE05], IEEE-1666 (current version 2.3.1), is an open-source design and modeling standard developed by the Accellera Systems Initiative [wwwa] for the design of hardware-software systems. Strictly speaking, SystemC is not a language, but a library of C++ with a set of coding rules and macros. However, it is usually described as a System-Level Design Language (SLDL).

SystemC models are hierarchical and executable. The models are composed by modules (called `SC_MODULE`) consisting of input/output ports, internal signals, concurrently running imperative processes and instances of other blocks. The execution of models is governed by the discrete-event (DE) driven simulation engine provided by SystemC.

In order to perform simulations, SystemC relies on the discrete notion of time with a configurable time granularity (from femtoseconds to seconds). This time might be used to provide a global notion of time among the components of SystemC models. Events are instantaneous in SystemC, and simulation time of processes is zero. Therefore, the processes triggered by a given event are executed sequentially, based on the concept of the 'delta-cycle'. The delta-cycle is basically an infinitesimal physical duration that does not advance simulation time, which is typically used to perform a sequential simulation of simultaneous tasks in zero simulation time, giving the illusion of a concurrent simulation of simultaneous processes.

One of the main strengths of SystemC is that hardware and software components can be described using a common language. What is more, both HW and SW components can be indistinguishable at the beginning of the design, since they are assigned to abstract modules that are only later refined as HW or SW components. Due to its capability to describe both HW and SW components in a unique language, and its ability to simulate concurrent processes, SystemC has nowadays become the de-facto standard in HW/SW system development.

Each module of SystemC can express a set of processes, via the `SC_METHOD()`, `SC_THREAD()` and `SC_CTHREAD()` commands. These processes are distinguished as explained in the following:

- `SC_METHOD`: Every time the simulation calls one of these processes, they run from the beginning to the end and they return. They are not allowed to suspend or to be interrupted, so they are considered as atomic functions.

- `SC_THREAD`: Processes of this type are started only once by the simulator. Once the thread starts to execute, it controls the simulation until a `wait()` statement is found, which suspends the simulation and gives the control back to the simulator. Hence, `SC_THREAD` processes usually contain an infinite loop containing one or more `wait()` statements. It is also possible to terminate the thread by using a `return` statement.

- `SC_CTHREAD`: This process type is a variant of the `SC_THREAD` process, with the variation that imposes the process to be sensitive to clock edges. In addition to `wait()` statements, `SC_CTHREAD` processes can use `wait_until()`, which is equivalent to repeat `wait()` functions in a loop until a certain condition holds.

In industry, it is infrequent that all modules within a system are modeled at the same level of abstraction simultaneously. Instead, commonly, different models of abstraction are required within a given system during its development, for example:

- A designer may use a very detailed model for a design under test but a very abstract model for the generation of the stimuli for the system.
- With a very detailed model as a starting point, the designer might create a more abstract model in order to increase simulation speed when testing another part of the system.

In order to deal with this issue, SystemC allows modeling systems at diverse levels of abstraction, and even enables designers to model subsystems of a given system at different abstraction levels, such as the register transfer level (RTL) or the more abstract transaction-level model (TLM) [Ghe06]. Hence, SystemC does not impose a top-down or a bottom-up design flow. Besides abstraction, SystemC also provides the other two simplification strategies presented in Section 2.3 in the development of the models:

- Abstraction: SystemC enables the designers to define hierarchical modules in order to hide or show the internal details of each component as desired.
- Partitioning: Communication and computation concerns are strictly decoupled in SystemC. Furthermore, partitioning may also be applied in the models by defining independent sub-modules for different aspects of the functionality of systems. The overall functionality is then achieved by the interaction of such sub-modules.
- Segmentation: SystemC enables simulating the resulting functionality of systems containing simultaneous sub-modules, by running them sequentially and making use of the concept of the delta-cycle.

Although the simulation engine of SystemC is natively event-triggered, different extensions have been made to it in order to support different MoCs. Some of them are briefly introduced in the following:

- SystemC-AMS [IEE13]: SystemC-AMS (SystemC - Analog Mixed Signal) is an extension to SystemC that enables the developers to model and simulate analog and mixed (analog and digital) signals, so that continuous-time models and discrete-continuous heterogeneous models can be simulated and verified.
- SystemC-H [PS04,PS06]: This extension to the SystemC language focuses on the design of heterogeneous models. It provides a simulation kernel that is capable of modeling and simulating parts of the system based on the concurrent sequential processes (CPS), finite state machine (FSM) and synchronous dataflow (SDF) MoCs.

- HetSC [HSV05, HV06, HV08, wwwl]: HetSC is another approach for the modeling of heterogeneous systems in SystemC. It is essentially a library of MoC-specific communication channels that can be introduced in SystemC models to simulate the behavior of systems based on different MoCs. Among others, the library includes communication channels for Kahn proccess networks, synchronous-reactive models or synchronous dataflow models.

- HetMoC [ZSJ10]: More recently, the KTH Royal institute of Technology developed a formal heterogeneous framework for the development of multi-MoC systems in SystemC. This framework enables designers to design a network of heterogeneous processes that communicate among themselves by the exchange of signals via FIFO channels. Different domains are integrated by the use of domain-specific interfaces. The HetMoC framework provides interfaces for SR, DE, continuous and untimed models of computation.

- SystemC-MDVP [FWI+14]: The SystemC Multi Domain Virtual Prototypes (SystemC-MDVP) is a language that is being defined in the CATENE CA701 H-INCEPTION project [wwwb]. The language is currently under development. It is being designed as an extension to SystemC-AMS. Therfore, analog-mixed signal simulation will be handled by SystemC-AMS, and other physical domains such as microfluidic systems will be supported by defining and integrating the necessary MoCs in the language. [FWI+14] shows some promising preliminary results of the capabilities of SystemC-MDVP.

CHAPTER 3

State of the Art

This chapter analyzes the state of the art in safety-critical embedded systems modeling languages and simulation frameworks, and discusses different approaches in simulated fault injection over different modeling languages.

From our knowledge and experience, a industrially successful modeling language/ framework for safety-critical real-time systems should at least include the following features:

- Provide a consistent notion of time: Time is a key concept in the design of embedded real-time systems [Per11]. Thus, the modeling language should rely on a timed MoC, thus making the temporal properties of systems explicit. This fact would ease their verification and validation process, and it would facilitate the deployment of such systems onto distributed platforms.
- Provide a time and value deterministic environment: A time and value deterministic simulation and execution environment eases the certification process of safety-critical systems by reducing their state space. Furthermore, such an environment enables repeatability of test-cases, a crucial property when it comes to prove that previous bugs in the system have been successfully fixed.
- Enable testing and simulated fault injection: Testing by simulation and injecting faults in the systems during these simulations enable carrying out validation and verification activities long before a prototype of the system is built. This way, the cost of fixing design bugs can be reduced by avoiding their propagation to the following stages of the development process. If possible, including formal verification techniques is also desirable.
- Provide techniques to tackle the complexity of systems: The increasing complexity of embedded and safety-critical systems makes their development a difficult

task. Thus, it is crucial reduce the complexity of their design, to facilitate both the design itself and the verification and validation activities. Mechanisms to support abstraction, partitioning, segmentation, and separation of concerns between functionality and platform-architecture are good candidates to be included in the modeling language.

According to these requirements, FTOS (see Section 3.4) and the E-TTM (see Section 3.5) are the most promising approaches for the modeling of safety-critical real-time systems. However, as explained in their corresponding sections, they still lack some of the features.

3.1 Model Driven Architecture (MDA)

As we explained in the previous chapter, the design of HW/SW systems is a complex and expensive task. Model Driven Engineering (MDE) approaches, such as the Model Driven Development (MDD) [Sel03], tackle this complexity by applying diverse techniques, such as raising the level of abstraction of their models in the first stages of the design. The MDA standard [MM03] is a specification for the implementation of MDD proposed by the Object Management Group, which tries to increase the portability, interoperability and reusability of models. MDA goes one step further on the definition of MDD by formalizing the development of systems in several steps and making use of the so-called “separation of concerns”, which suggests to separate the design of the system in two different models, the PIM and the PSM. First, the functionality of the system is defined as a PIM. Once the PIM has been defined and verified, it is transformed into a PSM by applying several platform specific refinements. The Y-chart development process introduced in the previous chapter can therefore be seen as a concretization of the MDA approach, where the refinement from PIM to PSM is done by defining a platform model (PM) and deploying PIM components into the elements of the PM.

MDA is intended to support model-driven engineering of software systems. The MDA is a specification that provides a set of guidelines for structuring designs expressed as models. Using the MDA methodology [MM03], system functionality may first be defined as a platform independent model (PIM) through an appropriate Domain Specific Language. Given a Platform Definition Model (PDM) corresponding to CORBA, .Net, the Web, etc., the PIM may then be translated to one or more platform specific models (PSMs) for the actual implementation, using different Domain Specific Languages, or a General Purpose Language like Java, C#, Python, etc. The translations between the PIM and PSMs are normally performed using automated tools, such as model transformation tools (e.g., tools compliant to the QVT standard). Although the MDA is a methodology for the development of systems, its principles have been successfully applied to other

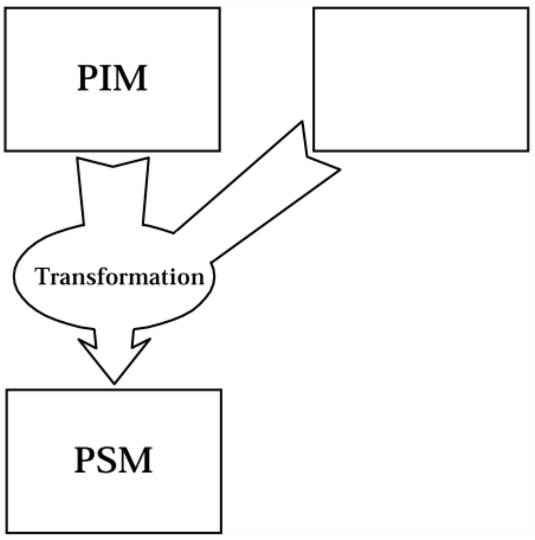


Figure 3.1: Model Transformations according to the MDA (from [MM03])

areas such as business process modeling, where the architecture and technology neutral PIM is mapped onto either system or manual processes.

Figure 3.1 summarizes the basic notion of the MDA. Regarding to this figure, the MDA Guide [MM03] sets the following: "The drawing is intended to be suggestive. The platform independent model and other information are combined by the transformation to produce a platform specific model. The drawing is also intended to be generic. There are many ways in which such a transformation may be done. However it is done, it produces, from a platform independent model, a model specific to a particular platform." Besides, the MDA Guide adds the following statement: "a PSM may provide more or less detail, depending on its purpose. A PSM will be an implementation, if it provides all the information needed to construct a system and to put it into operation, or it may act as a PIM that is used for further refinement to a PSM that can be directly implemented."

Both statements clearly show that the MDA leaves the specification of the topics to include in the PIMs and PSMs and the transformations between them undefined and open to any interpretation. Thus, it is a matter of each developing team to decide where to set the boundary between what should be part of the PIM and what should be introduced in the PSM.

This freedom in the choice of the concrete definition of PIMs and PSMs greatly increases the usability of the MDA. On the contrary, precisely this lack of precise definition hinders the generation of generic tools and automatic transformers/translators.

Besides these difficulties, the MDA is successfully established as a group of multiple standards released by the OMG, such as the Unified Modeling Language (UML) and the Systems Modeling Language (SysML). Note that the term "architecture" in Model-

Driven Architecture does not refer to the architecture of the system being modeled, but rather to the architecture of the various standards and model forms that serve as the technology basis for the MDA. These standards are described in the following sections.

Unified Modeling Language (UML)

The Unified Modeling Language standard (UML) [OMG11a] is probably the most representative of the standards for model-based software and systems development. It has had a significant success in the software industry as well as in other domains such as IT and financial systems. UML is a technique for software development that stresses a successive refinement approach to software design, and it is now the most widespread language used for modeling in both industry and academia.

UML was born when the emerging concept of graphical languages was fused with the object-oriented (OO) languages. The language has a broad scope and a structure based on graphical diagrams that allows the designers to customize and extend it to particular application domains. The original focus was on the graphical description of the architecture of software, i.e., the causal, communication, and hierarchy relationships.

Although it presents some difficulty in expressing constructs that are common to standard programming languages such as one-dimensional and sequential operations (assignments, loops, branches), and despite the scarcity of efficient code generators, the standard has been widely accepted by the industry. In fact, its graphical nature makes it a favorite for software developers to describe the structure of their programs.

UML initially targeted software development. However, it was designed as a foundation for generating different domain-specific languages, mainly through its profile mechanism. This capability allows the general concepts of UML to be specialized for a specific domain or application.

Regarding real-time systems design, two are the main weaknesses that UML presents. First, since plain UML targets software development, it misses the concept of time in its diagrams, and therefore concepts such as concurrency cannot be directly addressed. In fact, the semantics of the language are left unspecified except for the state diagrams, which have the semantics of state charts [JSEB04, SV07, BGP07]. As time is a central part of real-time systems, this greatly complicates the design of such systems in UML. Second, the diversity of diagrams makes it impossible to check that the overall description is consistent in the sense that the parts fit together appropriately. This difficulty gets even higher when the models make use of different profiles, which might overlap in some definitions or even present incompatibilities among them [SV07].

Systems Modeling Language (SysML)

Among the plethora of profiles of UML, SysML [OMG12] is of great potential interest for embedded system design.

SysML is an OMG standard language that adapts UML for systems engineering. Namely, it can be used “for specifying, analyzing, designing, verifying and validating complex systems that may include hardware, software, information, processes, personnel, and facilities” [OMG12].

The so-called “Block concept” of SysML, which abstracts the software details in UML classes, is a significant extension in the direction of modeling complex embedded real-time systems, where software is just one aspect besides electronics, mechanical parts, information units, and almost any other type of structural entity in the system of interest. Blocks can be used to decompose the system into individual parts, with dedicated ports for accessing their internals.

Blocks articulate a set of modeling perspectives enabling a separation of concerns during systems design. Similarly to UML, each of these perspectives is depicted in a dedicated diagram.

Due to its focus on system modeling, SysML adds requirements modeling as a key aspect of the system-development process. It provides requirements diagrams, tree structures and tables, which not only support the process of documenting the requirements, but also provide traceability to requirements throughout the design flow, ensuring that requirements are satisfied.

The four pillars of SysML are thus the capability of modeling of requirements, behavior, structure, and parameters [GKL+10, GETS10]:

- Requirement diagrams provide constructs for specifying text-based requirements and their relationships, including requirements hierarchies, as well as derivation, satisfaction, verification, and refinement relationships between requirements.

- Three types of diagrams model the behavior for the system: state-transition diagrams, activity diagrams and sequence diagrams. State-transition diagrams specify the behavior of the system as a finite series of discrete states. The system may transit from one state to another in case a certain event occurs. On the other hand, activity diagrams enable designers to specify the behavior as a linear sequence of transformations that convert a set of inputs into outputs according to some mathematical specification. Finally, sequence diagrams enable the SysML user to define the inter-relationship of the system with its external environment.

- Structure is represented by block-description diagrams and internal-block diagrams. Those diagrams enable the specification of more generic interactions and phenomena than those existing just in software systems. This includes physical flows such as liquids, energy, or electrical flows. The dimension and measurement units of the flowing physical quantities can be explicitly defined.

- Parametric diagrams allow the designers to describe, in a graphical manner, analytical relationships and constraints, such as those described by mathematical equations.

Similarly to UML, SysML also supports extensions for guarding the information flow and the entities of the system. SysML is an improvement over UML in that it allows to articulate requirements concerns relevant at the system engineering level, including function networks, and requirements allocation to subsystems. However, both UML and SysML lack the binding to a concrete system model that enables formal analysis of requirements and their associated models. Also, there is still too little support for a seamless transition between the requirements development and other development activities in SysML [KFFM10].

MARTE

As stated in previous descriptions, SysML and UML hardly formalize real-time aspects of embedded systems or at least not with the required rigor, what makes them unsuitable to model real-time systems. The UML profile MARTE (Modeling and Analysis of Real-time Embedded Systems) [OMG11b] is the UML standard extension to support the modeling of real-time embedded systems, and provides several mechanisms specified by the MDA to tackle the complexity challenge. Moreover, the support for real-time aspects opens the possibility for the generation of executable models by generating automatic code for SystemC models, as multiple research works have shown ([AH08, MMP09, PMPV10, PHV12]).

In MARTE, the time representation can be physical (dense or discretized) or logical (related to user-defined clocks) [AMS07]. The time structure of a design is built using time bases (TimeBase), a totally ordered set of dense or discretized instants, in which a distributed or multi-threaded application uses multiple time bases (MultipleTimeBase) which are a priori independent. The progress of physical time is measured using the model element clock, which references a given time base. The Clocked Constraint Specification Language (CCSL) enables semantic and graphical representation using stereotypes of clock constraints such as periodicity, coincidence, strict precedence, independence and exclusion [Kyu08].

MARTE provides an interesting foundation for the modeling of safety critical embedded systems [KAVS08], but this is still a challenge for different reasons such as [Per11]:

- **Consistency**: Ensuring consistency of the designed model properties and constraints up to the implementation and execution framework is also a key challenge.
    - **Formalism**: MARTE as UML is a semi-formal language, and this is why it imposes some limitations on the way functionality and time properties can be expressed and verified at model level and during the implementation process up to the (distributed) execution platform.

  - **Time determinism**: While MARTE supports the notion of time and the generation of executable SystemC models, it does not provide clear restrictions on handling simultaneity. Therefore, the execution of simultaneous components could lead to race conditions, because it does not naturally support temporal execution determinism.

- **Execution framework**: It does not restrict the execution framework, so designers must choose their desired execution target and restrict their design to that framework by themselves.

- **Distributed execution**: It does not restrict the execution topology, distributed or single node.

- **Codesign**: It does not restrict the implementation technology, which could be software, hardware or both.

3.2 Architecture, Analysis and Design Language (AADL)

The Architecture Analysis & Design Language (AADL) [Aer09] is both a textual and graphical language used to design and analyze software and hardware architectures of real-time systems. Standardized by the Society of Automotive Engineers (SAE), it was originally designed for the avionics industry. The main objective of AADL is to provide support for an early analysis of the properties of a system, which are supposed to be obtained before the coding or the deployment of the system.

The language enables the designers to describe their system in terms of software and hardware components. In addition, it allows them to specify how software components are mapped onto hardware components.

AADL includes all the standard concepts of an Architecture Description Language: components, connectors (used to describe the interface of components), and connections (used to link components). The language enumerates ten component categories divided into three groups to design the system:

- Software components:
  - data
  - subprogram
  - thread
  - thread group
  - process

- Hardware components:
  - memory
  - processor
  - bus
  - device
- System components:
  - system

SW models in AADL follow a hierarchical fashion, where the components are specified as follows: a process represents a virtual address space, or a partition; this address space includes the program defined by its sub-components. A process must contain at least one thread or thread group. A thread group is a logical organization of threads in a process. A thread represents a sequential flow of execution; it is the only AADL component that can be scheduled. A subprogram models a piece of code that can be called by a thread or another program. Data represents static variables used in the code; threads and processes can share data [BBC+09].

Besides the description of the functional properties of the system, AADL also covers the specification of the non-functional properties. To this end, AADL defines the notion of properties that can be attached to most elements, similarly to MARTE. Properties are attributes that specify constraints or characteristics that apply to the elements of the architecture, e.g., clock frequency of a processor, execution time of a thread, bandwidth of a bus, etc. Typical properties of components are already defined by the standard. However, the set of properties might be extended with user-defined ones.

In addition, the AADL standard contains a number of annex libraries that define extensions to the core language concepts. The Error Model Annex [Aer11a] is maybe the most significant annex of the standard. This report allows the definition of error models associated to hardware, software or system components, consisting of error state definitions, error propagations and their probability.

Similarly to MARTE, AADL itself defines no specific model of computation [BKSZN10]. Thus, it is the responsibility of the designer to define the system in such a way that the desired temporal constraints are guaranteed, using the event-related mechanisms provided by the language. Threads are the only components that have an execution semantics. AADL supports the classical types of dispatch protocols: a thread can be declared as periodic, aperiodic, sporadic or background. Threads have two predeclared event ports: dispatch and complete. The dispatch port is used for aperiodic or sporadic threads. If this port is connected all other ports of the thread do not trigger the dispatch. It is a very common behavior for an aperiodic or a sporadic thread to send an event on completion. The complete event port is used to send an event at the end of the execution [BBC$^+$09].

In order to enrich AADL with an implicit model of computation, and that way give the standard the capability to support formal analysis, verification and synthesis of timed properties, [BBD$^+$14] suggests to equip the standard with a synchronous model of computation. To do so, the research work defines an algebraic framework in which time is formally defined from implicit AADL concepts.

3.3 Distributed Application Layer (DAL)

The Distributed Application Layer (DAL) [SBR$^+$12, wwwf] is a scenario-based design flow for mapping a set of applications onto heterogeneous on-chip many-core systems. DAL has been developed in the European Reference Tiled Architecture Experiment (EURETILE) project [wwwi], as a continuation of its predecessor, the Distributed Operation Layer (DOL) [BBH$^+$07, wwwg], and heavily relies on it.

The Distributed Operation Layer (DOL) [BBH$^+$07, wwwg] is a design flow for the model-driven development [Sel03] of multiprocessor streaming applications. DOL was developed by the Computer Engineering Group of the Swiss Federal Institute of Technology Zurich (ETH) in the context of a European Framework Programme 6 research project called "Scalable Software/Hardware Architecture Platform for Embedded Systems" (Shapes) [SBF$^+$07].

The DOL design flow basically follows the Y-chart paradigm introduced in section 2.4, and takes advantage of the orthogonalization of concerns provided by the mentioned paradigm to separate the definition of the SW and HW architectures and the mapping between both of them. According to this, the design of a system in DOL is performed in three independent tasks:

- Specification of the application: specifies the functionality of an application, i.e., the function of individual actors and the topology of the network.
- Specification of the architecture: captures the key characteristics of an architecture and the run-time environment running on top of it as seen from a software developer's viewpoint, that is, the available processors, communication drivers, and memories.
- Specification of the mapping: determines the spatial and temporal mapping (binding and scheduling) of the application onto the architecture, that is, the mapping of actors onto processors, and the mapping of the FIFO channels to communication drivers.

Once the specifications are defined, DOL suggests to automatically implement the system using software-synthesis tools, and quantitatively evaluate its performance in order to modify/improve any of the specifications in case the behavior of the system is unsatisfactory.

The performance evaluation is made by simulating the models. Executable DOL models rely on the Kahn Process Networks MoC. A KPN basically specifies an application as a set of autonomous processes that communicate through point-to-point FIFO channels. To represent the interactions between the applications, a finite state machine is used, where each state represents an execution scenario, i.e., a set of running or paused applications.

DOL was designed as a textual language, defined as an XML-based specification format. In order to increase its usability, a graphical front-end called “Moses” was developed at ETH [wwwq].

DAL was developed as an improvement over DOL for the development of software systems in multi- and many-core platforms. It supports the design, the optimization, and the simultaneous execution of multiple dynamically interacting streaming applications on heterogeneous platforms. Compared to DOL, DAL is intended for the design of dynamic applications, and therefore focuses on automatic remapping, system-level analysis and fault recovery.

Similarly to DOL, applications in DAL are specified as executable Kahn Proccess Networks. In addition, DAL enables to capture the dynamics of the system as a set of scenarios in a finite state machine (FSM), where each scenario, identified as a state of the FSM, represents a set of concurrently running or paused applications.

The specification of DAL models is also carried out in textual format, compliant with the XML-based syntax determined by the DAL. Additionally, *DALipse* [wwwe], a visual editor developed as an Eclipse plugin is available to specify DAL applications. DALipse enables the programmer to visually specify finite state machines and process networks, along with the ability to optimize their mappings, to functionally test the system by simulation, and to generate code for one of the supported target platforms.

3.4 Fault-Tolerant Operating System (FTOS)

The Fault-Tolerant Operating System (FTOS) [BKSZN10, Buc08], is a model-based development tool created at Technical University München (TUM) for the design of dependable automotive systems. It supports modeling of dependable systems and code generation for non-functional properties such as scheduling, communication within the distributed system, and fault-tolerance mechanisms.

The FTOS shows a special interest in the non-functional requirements of dependable systems, and highlights the idea that the fault-tolerance mechanisms are generated tailored for the application and the target hardware. In fact, unlike the other approaches reviewed before, the FTOS suggests to divide the definition of the system into four different models: the HW architecture model, the SW architecture model, the fault model

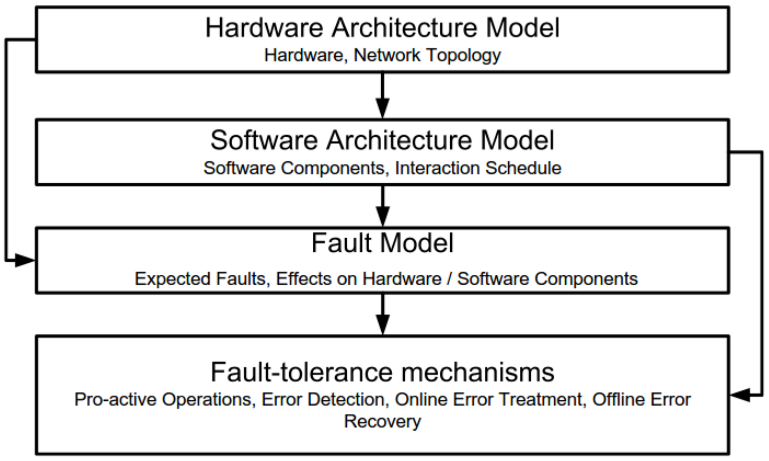

Hardware Architecture Model
Hardware, Network Topology
Software Architecture Model
Software Components, Interaction Schedule
Fault Model
Expected Faults, Effects on Hardware / Software Components
Fault-tolerance mechanisms
Pro-active Operations, Error Detection, Online Error Treatment, Offline Error Recovery

Figure 3.2: Models of systems in FTOS and their dependencies (from [Buc08])

and the fault-tolerance mechanisms. Figure 3.2 illustrates the dependencies between the models specified in FTOS.

As Figure 3.2 shows, the definition of the system in FTOS starts with the specification of the HW architecture model, which describes the relevant information about the hardware components and the network topology.

When HW related aspects have been defined, the designer can specify the software architecture model. This model, and more especially its model of computation play a central role in FTOS. According to the designers, "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, what raises their complexity" [Buc08]. Therefore, in order to reduce the cognitive complexity of the systems as much as possible, the SW model in FTOS raises the abstraction level of the temporal specification and provides a time deterministic environment relying on the Logical Execution Time MoC ([KS12], section 2.5). Moreover, by reducing the complexity of the system, this design decision enables to simplify the fault-tolerance mechanisms.

Once the HW and SW of the system have been defined, the designer should think about the fault-tolerance mechanisms the dependable system under design should include. However, the selection and implementation of such mechanisms depends on the probability, type and location of the possible faults. Therefore, the development of the system continues by the development of its fault model. During the fault model design, the FTOS relies on a number of generic fault effects for each software/hardware component type, and forces the developer to define all the fault assumptions, including the Fault Containment Regions (FCRs), effects of assumed faults, fault configurations to be considered and activation assumptions. As the fault effects specified in the fault model

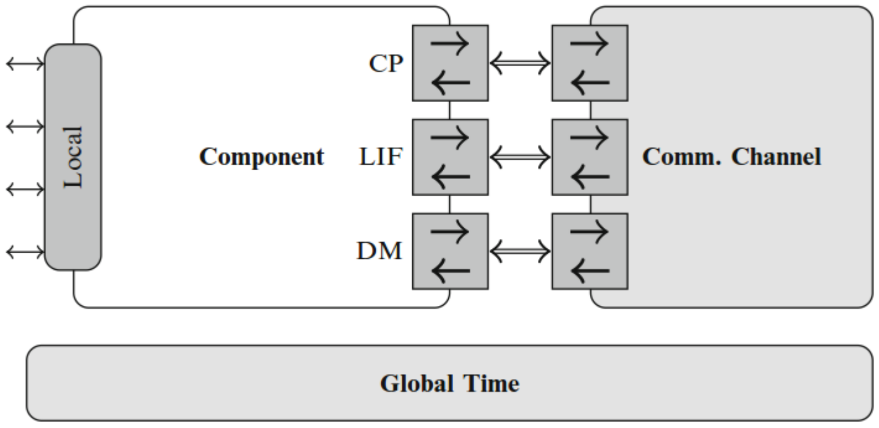


Figure 3.3: E-TTM elements and relationships (from [PNOES10a])

are generic, their respective fault detectors can also be generic.

Finally, the FTOS finishes the definition of systems by the description of the fault-tolerance mechanisms. The fault-tolerance concept in FTOS consists of the definition of a fault-tolerance mechanism for each possible error. Four different types of mechanisms are defined to do so: proactive operations, error detection, online error treatment and offline error recovery.

3.5 Executable Time-Triggered Model (E-TTM)

The E-TTM [PNOES10b, PNOES10a, PONA11] is a SystemC based extension for the modeling of dependable real-time embedded systems based on the TTA. It extends SystemC with the sparse notion of time (see Figure 2.8d), and decouples computation activities from communication, while they both share a global notion of time.

The E-TTM focuses on the description of the functionality of embedded systems deployed onto TTA platforms. Following the definition provided by the TTA paradigm [KB03, Kop08], components in E-TTM models are self contained subsystems that can be used as building blocks. Thus, the E-TTM enables the designers to define the system in a hierarchical fashion, by composing it from components.

As shown in Figure 3.3 E-TTM is based on a strict separation of concerns (partitioning) between computation and communication, while the global notion of time is common for both. Components communicate with each other by the exchange of messages across ports connected to communication channels that provide interfaces.

In E-TTM models, components are classified in computational components and interface components. Interface components act as gateways whereas computational components accept input messages, provide a useful service (computation) and produce output messages after some time. Depending on their activation mechanism, computational components can be defined as time-triggered or event-triggered. Time-triggered components are triggered by clock events, which always take place during the sparse-time activity intervals. Event-triggered components are activated by external events. External events can occur at any instant of the continuous time, and are therefore not compliant with the sparse model of time. Hence, the E-TTM interface delays the triggering of events until the next activity interval. This way, the E-TTM guarantees that the executions of components always take place at the activity intervals of the sparse model of time.

From an architectural point of view, computational components follow the specification of the TTA. They are consequently divided into *systems*, *Distributed Application Subsystems (DASes)*, and *jobs* [PNOES10a]. The former two are hierarchical components that might be composed by sub-components, whereas the latter are atomic (see Section 4.5).

The execution of computational components is instantaneous and it never takes place between macroticks, during the silence-interval. Although simultaneously triggered components are executed sequentially, the output messages are not delivered during the execution macrotick and as previously stated, the external events generated during the computation do not trigger activation events until the next execution macrotick. This guarantees that the execution order chosen by the E-TTM scheduler for simultaneously triggered jobs does not have any impact on the outputs of the system.

The components communicate with each other by exchanging messages across ports. As figure 3.3 illustrates, computational components contain the four interfaces specified by the TTA ([Kop11], Section 2.6), i.e., Linking Interface (LIF), Configuration and Planning Interface (CP), Diagnostic and Maintenance Interface (DM), and local interfaces. Components send messages during the activity interval to the communication channel, which handles the exchange of messages with other components. The communication channel delivers all sent messages to destination component ports during the silence-interval, a delta-delay after the activity-interval, in order to ensure determinism in the system. The exchange of messages is performed by the communication infrastructure quasi-instantaneously. However, E-TTM enables the designer to delay the transmission of a message in order to represent the amount of time that computation or message-exchange take in the real world.

Since the E-TTM is focused on modeling time-triggered dependable systems, it strictly limits its models to rely on time-triggered models of computation, which might limit its applicability for the design of heterogeneous cyber-physical systems. However, the fact that E-TTM has been built as an extension to the SystemC language eases its integration with other models relying on different models of computation. This way, the time-triggered components of a heterogeneous cyber-physical system can be modeled in SystemC using the E-TTM, which would guarantee that the time-triggered properties of the component are not violated. At the same time, the rest of the (non-time-triggered) components could be modeled with pure SystemC or a suitable extension (see Section 2.7), depending on the MoC they rely on.

The E-TTM is restricted to the modeling of the functionality of dependable time-triggered models. Therefore, the specification of the HW related aspects of the system would require an extension of the E-TTM, in order to be able to model platform specific aspects, such as the properties of the target HW platform.

3.6 Simulated Fault Injection in VHDL Models

Injecting faults into systems during simulation is a straightforward technique to verify that such faults do not cause failures in the system, i.e., the system is tolerant to those specific faults.

Thus, fault injection strategies and techniques have been very widely analyzed in the past [BP03, ZAV04] and several tools have been developed by the research community, most of them focusing on VHDL models. This section summarizes the most significant approaches.

MEFISTO [JAR$^+$94] (Multi-level Error/Fault Injection Simulation Tool), a simulated fault injection tool developed jointly by LAAS-CNRS and Chalmers University of Technology, was one of the earliest tools to conduct fault injection experiments using VHDL simulation models. In fact, it was the first tool to implement simulator commands, a technique that enables testing teams to use commands of the simulator to modify the value of the signals and variables of the model during simulation.

The research on MEFISTO evolved into two specific prototype instances developed and maintained by each institution, namely, MEFISTO-L (at LAAS-CNRS) [BPC98] and MEFISTO-C (at Chalmers University of Technology) [FSK98]. MEFISTO-L focuses on evaluating the fault-tolerance mechanisms of the designs by adding dedicated FI components to them, such as probes and saboteurs. On the contrary, MEFISTO-C tried to improve the efficiency of the original MEFISTO, and focuses on injecting faults via built-in simulator commands in variables and signals defined in the VHDL model. In order to improve the usability of the tool, it also offers users a variety of predefined fault models.

VERIFY [STB96,STB97] was a tool developed at University of Erlangen-Nurnberg. VERIFY modifies and extends the VHDL language with mechanisms for describing the possible faults that may occur at a specific component. This way, hardware manufacturers are enabled to express their knowledge about the fault behavior of their components directly in the VHDL model. For simulation, VERIFY implements a multi-threaded fault injection technique with checkpoints, which compares the faulty threads with a golden model in order to detect anomalies in the models.

SINJECT [ZME03], developed at Sharif University of Technology, provides a mixed-mode fault injection for both Verilog and VHDL models. The tool tries to overcome some of the limitations of the VHDL language, such as its incapability to simulate switch-level abstraction models and its limitations on fault modeling. To handle these limitations, it provides a fault injection environment for systems including parts modeled in Verilog and parts in VHDL, via the ModelSim simulator. SINJECT covers all levels of abstraction, from system-level to switch-level, for both functional and structural models.

Unfortunately, most of the fault injection tools for VHDL models suffer from the same drawback: the high temporal costs of their simulation. Obviously, this temporal drawback gets even worse when the system is complex, or is modeled at a low abstraction level. The most typical technique to cope with this situation is to raise the abstraction level of the models, or even switch to a system-level modeling language, such as System-Verilog or SystemC.

However, a more recent approach called FuSE [JDR09], developed by the Vienna University of Technology, suggests to cope with this problem by combining simulation and emulation based fault injection. The main advantage of the FuSE tool is that it supports emulation- and simulation-based fault injection but keeps the switching between these modes completely transparent to the user. This way, the speed-up provided by the emulation mode allows to perform a huge number of fault injection experiments within a reasonable amount of time, and the simulation mode offers the flexibility and visibility required for specific cases.

3.7 Simulated Fault Injection in SystemC Models

Although simulated fault injection in VHDL models has been a more prominent research field in the past, the fact that SystemC has nowadays become the de-facto standard in industrial HW/SW system design has contributed to an increasing interest on SFI in SystemC models. This section summarizes the most promising research works in the field of simulated fault injection based on SystemC models.

Misera et al. [MVS07] adapt fault injection techniques and strategies from VHDL models to SystemC models in order to analyze the limitations and possibilities of the SystemC kernel. They simulate systems including saboteurs and simulator commands, and they extend logic types of SystemC in order to perform a more realistic behavior of logic components. Since the approach is based on strategies used in VHDL models, they focus on logic-level models. In [SRAH08] Shafik et al. propose an alternative technique to the one presented in [MVS07], also focusing on logic-level models.

Bolchini et al. go one step further into multiple abstraction level fault injection in [BMS08]. The paper presents a fault injection environment for the ReSP simulation platform [BBF$^+$08]. The approach enables injecting faults by using saboteurs and simulator commands in components defined in the ReSP platform, using a new technique called reflective wrapper. The reflective wrapper technique inserts a Python [wwwu] layer between the SystemC kernel and SystemC IPs, which provides access to any SystemC element. Faults can then be injected into these elements either by parsing an XML file or directly typing commands in the console. Injection of faults in this approach is limited to elements defined as *public* objects, which might require some re-design of the SUT. The approach does not focus on a specific MoC, so simulation is paused and resumed whenever a fault is injected.

In [LR11] Weiyun and Radetzki use the Concurrent and Comparative Simulation (CCS) technique to inject faults in SystemC models. The CCS is implemented by extending data types to enable a differential representation of signal/variable values, which are implemented as lists. The first element of the sorted list corresponds to the non-faulty signal/variable value, while all other values in the list correspond to faulty values. When an operation involving a signal/variable with extended data type is made, the operation is executed for all the values in the list. This way faults are propagated all over the system. This approach makes it possible to perform more than one fault injection experiment in each execution. The main limitations of this approach are that the developer must use a specific data type in order to inject faults in variables, and fault libraries are not defined, so the tester must implement the fault models.

Reiter et al. [RPV$^+$13] perform error injection in simulated HW models. First virtual prototypes of the platform components are created using the CHESS modeling language, and then the models are extended in order to inject errors in them. The approach provides a library of different error models, including data-corruption, timing-corruption, halt, and signal-loss. The framework does not rely on a concrete model of computation, and the paper does not describe how timing constraints of the system under test are guaranteed.

CHAPTER 4

PS-TTM

This chapter describes the proposed Platform Specific Time-Triggered Model (PS-TTM), a modeling and simulation framework for the design of time-triggered safety-critical embedded systems at different abstraction levels, based on the Y-chart development process and the MDA approach in SystemC.

4.1 Introduction

The PS-TTM is a modeling and simulation framework for the development of Time-Triggered Architecture (TTA) based safety-critical embedded systems. The approach relies on a strict separation between the designs of the functionality of the systems and their target platform. Furthermore, the notions of computation and communication are decoupled, in order to ease the design of such systems.

The PS-TTM provides a value and time-domain deterministic simulation environment, which enables an early functional and temporal assessment of systems. Moreover, as will be further explained in chapter 5, the framework natively includes a mechanism to perform non-intrusive simulated fault injection (SFI) within the models. This SFI technique enables developers to carry out an early dependability assessment, which reduces the risk of a late and expensive discovery of safety related pitfalls.

In order to tackle the increasing complexity of current embedded systems introduced in Section 2.3, the PS-TTM enables designers to apply abstraction, partitioning and segmentation to their models. In addition, the time determinism provided by the PS-TTM simulation engine contributes to increase the simplicity of the models.

Following the approach presented in [Kop08], PS-TTM models are composed from computational components (c-components) and interface components (i-components). C-components get input data, perform some computation and produce output data, whereas i-components behave as gateways. With the aim to facilitate the modeling of TTA-based systems, c-components are time-triggered in PS-TTM, i.e., they are triggered by the occurrence of clock events that rely on a global sparse notion of time.

It is worth mentioning that the name of the framework might be misleading: as the following sections show, the PS-TTM must not necessarily be used to develop platform specific models of systems, but also more abstract platform independent models. However, this name has been chosen due to the capability of the framework to describe platform specific and mixed platform independent and specific models.

4.2 Overall Work-Flow

The PS-TTM modeling framework enables engineers to design their time-triggered systems following the Y-chart approach. Figure 4.1 shows a general overview of the development process suggested by the PS-TTM.

The development process starts with the definition of the functional and non-functional requirements of the system. This step is usually performed by the customers together with the developers. Once the requirements have been fully defined, the engineers can start with the design of the Platform Independent Model of the system. Simultaneously, the testing team begins the definition of the test cases and simulated fault injection campaigns.

When a functional model (PIM) of the system is ready, it is assessed by simulating the model against the test cases and the simulated fault injection campaigns defined by the testing team. The PS-TTM simulator provides specific mechanisms to simulate faults within the system. The result of the simulations are analyzed by the testing team. In case the simulations show any undesired behavior of the model, the engineers are notified about the bug and the PIM is redesigned. This loop is repeated until the model successfully passes all test cases.

During this process, another engineering team designs a platform model (PM) that fulfills the non-functional requirements of the system. When the PIM is considered satisfactory, the deployment step is started. This activity consists on deploying the platform independent model onto the platform, thus building the platform specific model (PSM).

The PSM is then simulated against the functional and non-functional test cases and simulated fault injection campaigns defined by the testing teams. The results of the simulations are analyzed, and in case any unwanted behavior is detected, the corresponding bugs are fixed in the PIM, the PM or the PSM. Once the PSM passes all test cases, the model is considered satisfactory and it is ready to start the prototyping phase.

This decoupling between the design of functionality and the design of the platform reduces design time and cost, and supports the early assessment of the system since a preliminary simulation and analysis phase is included in the process. Furthermore,

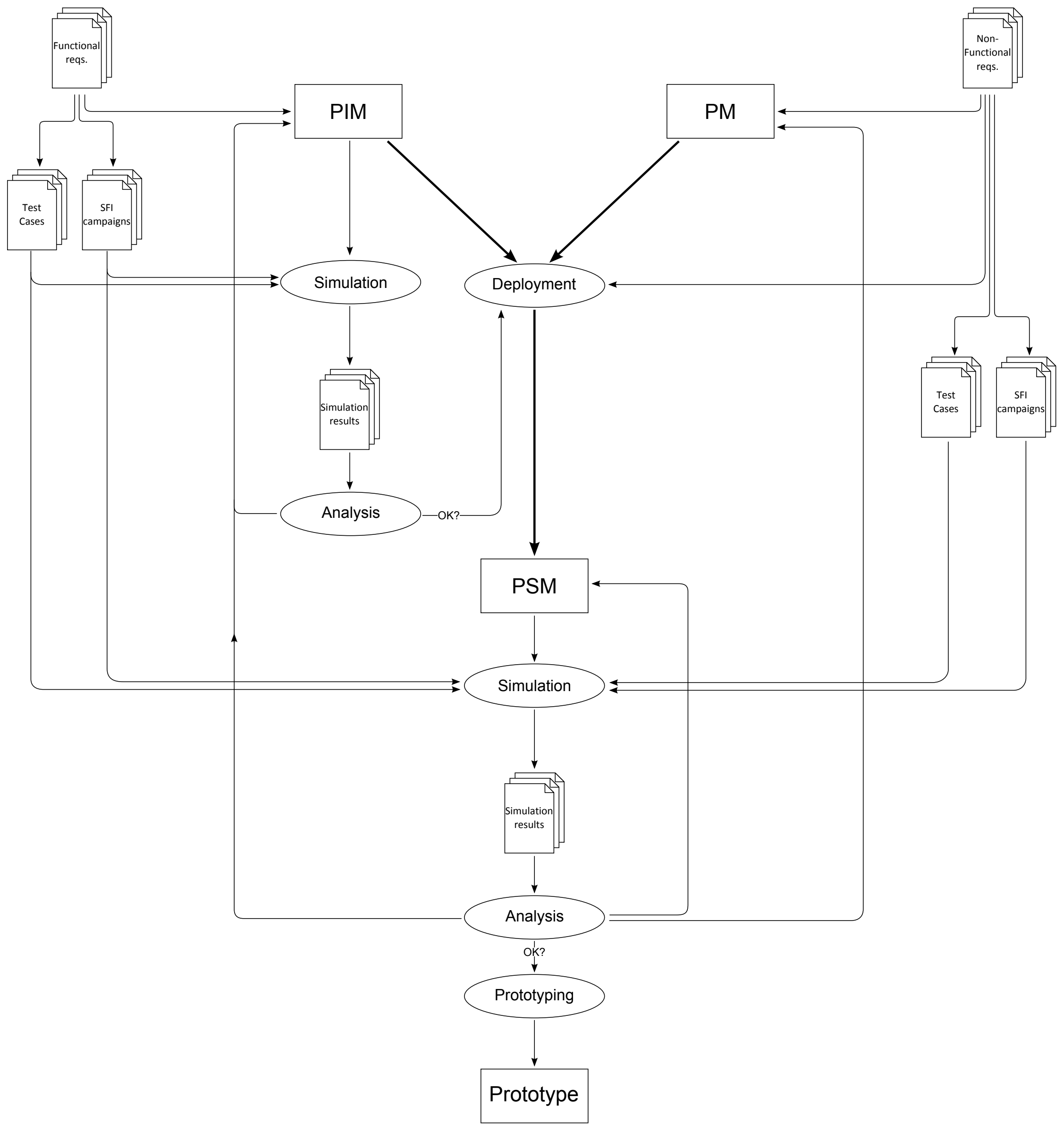


Figure 4.1: Overall work-flow with the PS-TTM

it eases the assessment of the emerging behavior of a given functionality in several platform variants, which facilitates the comparison of diverse design approaches in order to choose the most suitable platform.

4.3 The Meta-Model

The composition of the PS-TTM is illustrated by Figure 4.2. The PS-TTM modeling approach is defined as a meta-model, from which models can be derived, as the following sections will describe.

The components in light gray of the figure are those that are out of the scope of this work, but are used by the PS-TTM, namely, SystemC and the E-TTM. The E-TTM meta-model is compliant with SystemC. In fact, it enables the design of time-triggered safety-critical systems using SystemC. Similarly, the E-TTM execution framework makes use of the simulation engine provided by the SystemC library.

Components in white represent the core work of this thesis. The PS-TTM meta-model is compliant with the E-TTM. In fact, it has been defined as an extension to it. Besides, the PS-TTM implements two different simulation engines. The so-called PS-TTM execution framework (or PS-TTM simulation engine), enables the test engineers to carry out the simulation of PS-TTM compliant platform specific models, and it is based on the E-TTM execution framework. On the other hand, the PI-TTM execution framework (or PI-TTM simulation engine) provides the mechanisms to execute platform independent models, and makes use of the PS-TTM simulation engine. These two engines have been designed separately because PIMs and PSMs rely on different MoCs in PS-TTM, as Section 4.5 describes.

Finally, the components depicted in dark gray represent models of time-triggered safety-critical embedded systems developed by means of the PS-TTM, namely the PIM and the PSM, and must obviously be compliant with the PS-TTM meta-model. The framework enables developers to execute their platform independent and platform specific models by means of the PI-TTM and PS-TTM simulation engines respectively in order to evaluate their behavior under specified conditions.

4.4 Characteristics of the PS-TTM

The PS-TTM is built as an extension of the E-TTM library [PNOES10a, PONA11] that enables the engineers to model both PIMs and PSMs, and provides mechanisms to perform non-intrusive simulated fault injection on the models.

In order to make the design of time-triggered architecture based systems straightforward, the components of the PS-TTM are defined following the description given by Kopetz et al. [KS03, KOESH07], i.e., they contain four different interfaces:

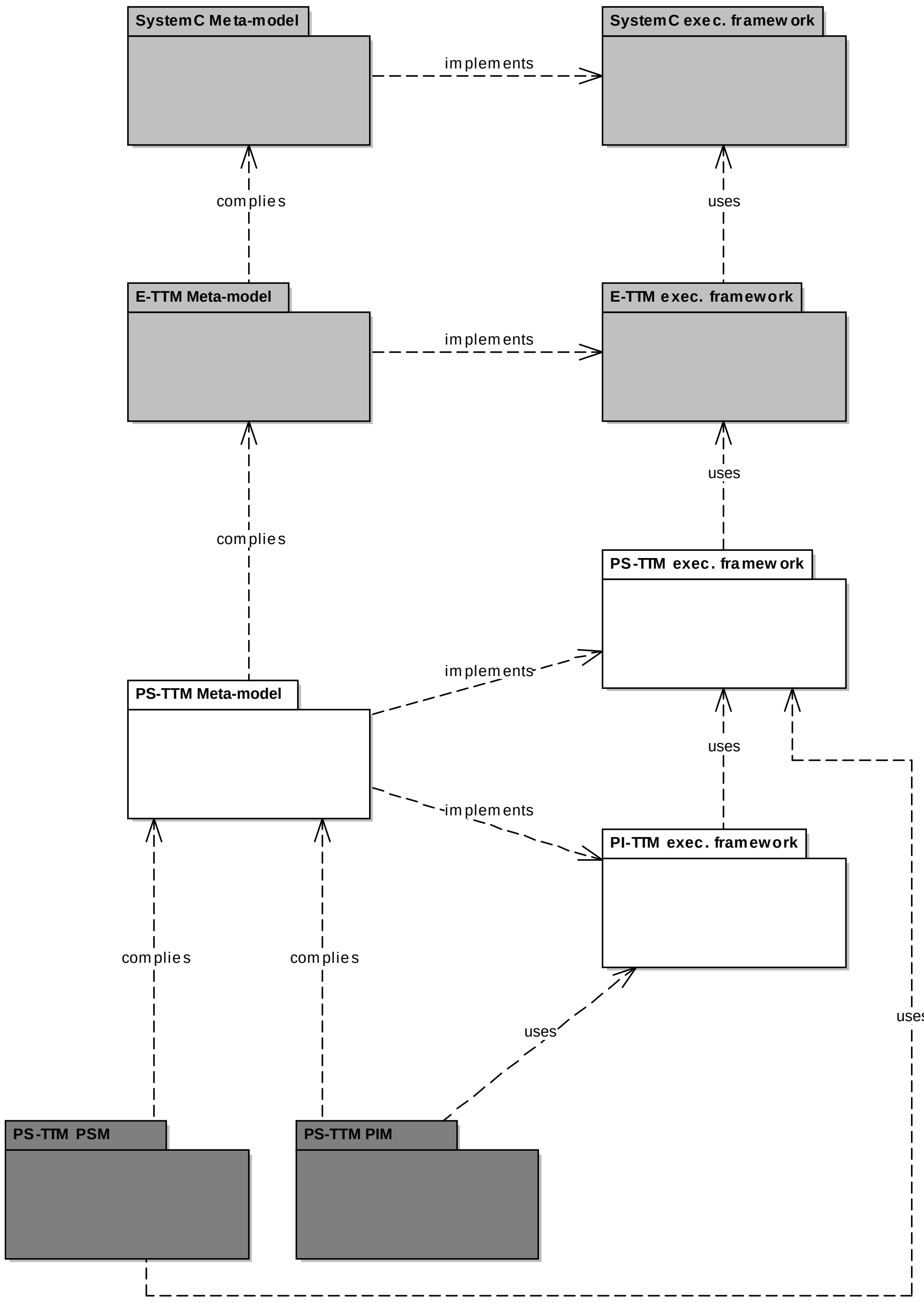


Figure 4.2: PS-TTM meta-model, execution framework and models

- **Linking Interface (LIF)**: The LIF is the interface that provides the timely information to the nodes during the operation of the system. This interface is used by the nodes to communicate with each other, and it is therefore a time-critical interface that must meet the temporal specification of the application in all possible scenarios.
- **Configuration and Planning Interface (CP)**: The CP is the interface used to configure the system, i.e., to connect a node to other nodes.
- **Diagnostic and Management Interface (DM)**: The DM is the interface that enables the maintenance engineers to observe the internal state of the node.
- **Local interfaces**: Local interfaces connect the component to the external world, i.e., the environment.

Computational components

Computational components are classified in two groups in the PS-TTM: hierarchical components (h-components) and atomic components.

Hierarchical components are basically composed by four parts:

- **Encapsulating boudary**: Defines the boundary between the h-component and the rest of the model. It provides the LIF, CP, DM and local interfaces for communication.
- **Set of c-components**: The functionality of a h-component is defined by a set of c-components that establish its behavior. Each c-component provides a chunk of the overall functionality of the hierarchical component. Therefore, in order to get a functional h-component, the minimum number of internal c-components is one.
- **Communication channel**: The communication channel connects all c-components inside the h-component and enables them to communicate with each other by pushing and pulling RT data. The communication channel shares the global notion of time with the rest of the system.
- **i-component**: The additional interface component enables the h-component to communicate with the rest of the components through the LIF, CP and DM interfaces, whereas it hides the internal details of the h-component to the others.

The encapsulating boundary, the communication channel and the i-component are automatically provided by the PS-TTM when a h-component is defined. Obviously, the internal set of c-components is specific to each h-component and must therefore be defined by the developer in order to get the desired model.

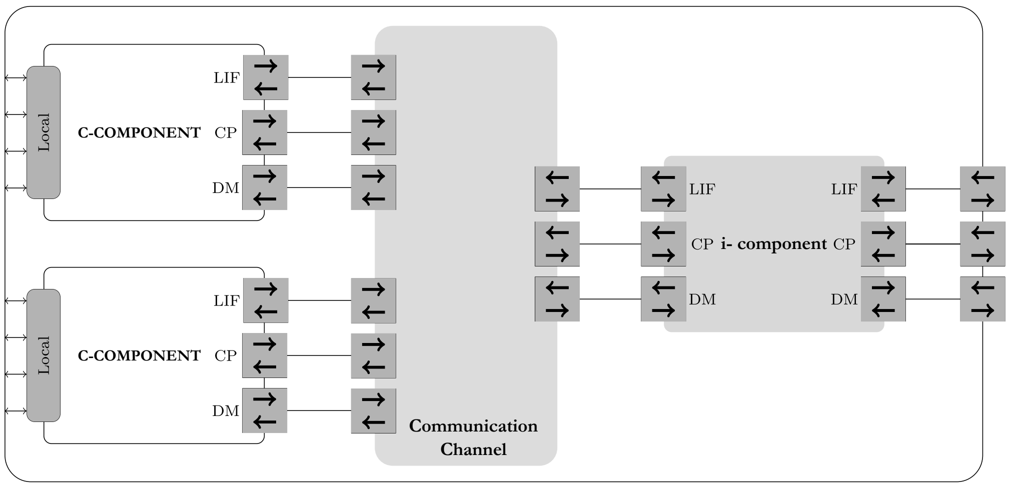


Figure 4.3: PS-TTM: architecture of hierarchical components

Figure 4.3 shows the architecture of hierarchical components in PS-TTM. The elements shaded in gray are provided by the modeling framework. The big box that wraps all the components represents the encapsulating boundary.

Atomic components are composed by:

- **Encapsulating boudary**: Defines the boundary between the h-component and the rest of the model. It provides the LIF, CP, DM and local interfaces for communication.
- **Functional task**: The functional task of an atomic component is the piece of software that provides it the desired functionality. It is defined in C/C++ language, and consists of a task that gets input data from the well known interfaces, performs some computation and gives back some output data to the system.

Figure 4.4 shows the structure of atomic components in PS-TTM. The external boundary represents the encapsulating boundary of the component.

Notice that internal c-components of the generic definition of hierarchical components (Figure 4.3) are not specified as atomic neither as hierarchical. In fact, in a specific instantiation, they could be any of them. If they were hierarchical, the architecture depicted in Figure 4.3 would be replicated inside the c-component. If they were atomic, the internal structure of the c-components would be the one depicted in Figure 4.4.

Communication

The PS-TTM provides a strict separation between computation and communication concerns. Following the time-triggered paradigm, communication is managed by a tempo-

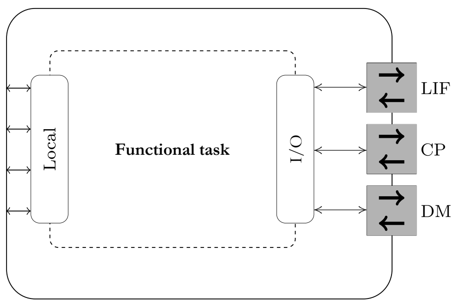


Figure 4.4: PS-TTM: architecture of atomic components

ral firewall [KN97], called *communication channel*. As figure 4.3 shows, each hierarchical component contains a communication channel, which is automatically created by the PS-TTM when a new h-component is defined.

The communication channel works as an intermediate storage memory for the c-components of the hierarchical system where they are located. It uses a combination of push-pull communication models, and consequently provides two different interfaces for communication:

- ***Push***: Enables the c-components to overwrite a given real-time entity image in the shared real-time entity of the communication channel.

- ***Pull***: Enables the c-components to get a given real-time entity image from the shared real-time entity of the communication channel.

In other words, whenever a c-component wants to submit some information, it can send data to the communication channel by using the *push()* command, whereas a c-component that needs to get any information can read data from the communication channel by calling the *pull* command.

Real-time entities of the communication channel keep their value until they are overwritten. The action of updating real-time entities is only governed by the communication channel itself, and can be delayed by an amount of time. Therefore, race conditions are avoided by design. This way, communication between components is allowed, whereas the designers have the guarantee that it will never occur an interruption that may disturb the temporal behavior of any component. The propagation real-time entities between different communication channels is carried out by the i-component. This propagation is uni-directional for each real-time entity, i.e., the real-time component is treated either as input or output at each hierarchical component. In order to avoid indeterminism, real-time components must work on a single-writer multiple-reader fashion, that is, only one atomic component is allowed to overwrite the value of the real-time entity of its hierarchical component.

The detailed functionality of the communication channel at each design level (PIM and PSM) is further described in the corresponding parts of section 4.5.

Tackling the complexity challenge

The complexity challenge introduced in section 2.3 is tackled by providing a global notion of time throughout the system and enabling to apply abstraction, partitioning and segmentation to the models.

In contrast to E-TTM, all computational components in the PS-TTM are triggered by time, according to the global notion of time provided by the PS-TTM infrastructure for all the components within a model. This fact, together with the time deterministic simulation environment provided by the PS-TTM, increases the timing predictability of the systems, which makes it well-suited for safety-critical embedded systems.

Abstraction is provided by the capability of the PS-TTM framework to define hierarchical systems. Hierarchy is achieved by the usage of h-components as c-components, which are further refined in a set of c-components, thus establishing a complete system defined over several abstraction levels. This hierarchical design is shown in Figure 4.3. Since each h-component must have at least one c-component inside, the lowest level of each hierarchical branch is always an atomic job, which provides the most basic chunk of functionality to the model.

Partitioning is enabled in two ways:

1. Since h-components enable to divide the overall functionality in different internal c-components, the system may be partitioned into quasi-independent components, each of them performing different parts of the overall functionality. An example of this is shown in Figure 4.3, where two c-components are placed inside the hierarchical component.

2. By design, the PS-TTM separates computation and communication concerns in different parts of the system. Computation is performed by c-components, whereas communication is handled by communication channels and i-components.

Finally, the time and value deterministic simulation engine included in the PS-TTM provides segmentation capabilities by automatically setting the sequential order of execution of the simultaneous jobs that may appear in a model.

4.5 Modeling at Different Stages of the Development

The following sections describe in detail the characteristics of the platform independent and platform specific models in PS-TTM.

Platform Independent Models

As introduced in section 4.2, the development of safety-critical time-triggered embedded systems in PS-TTM starts with the design of a Platform Independent Model (PIM) of the system. The PIM is derived from the functional requirements provided by the customer, so it is a purely functional model that captures the intended behavior of the system without taking into account its non-functional requirements.

In contrast to other approaches, PIMs are executable in PS-TTM, and they therefore allow the designers to check the logic of an application at an early stage of development without being concerned about all the constraints of the target environment. This approach may potentially save design time and cost during the development process, since the early validation capability it provides may facilitate the early detection of functional design bugs that would otherwise require an expensive redesign of the system.

The following sections describe in detail the fundamentals of the platform independent models in the PS-TTM.

Underlying Model of Computation

As explained in section 2.5, the underlying model of computation of a system can dramatically increase or reduce its cognitive complexity. Therefore, the choice of an adequate MoC for the design of a given system is crucial.

As pointed out by Buckl [Buc08], especially in the context of fault-tolerant systems, where different experts are involved in the development of the system and the internals of the systems must be very well understood, a simple execution model is essential.

Hence, in order to make the deployment of PIM models into TTA platforms straightforward while preserving the simplicity of the functional models, the PS-TTM relies on the LET MoC for the definition of platform independent models. The LET MoC defines the functionality of systems by specifying a logical duration for each computational job, regardless of its physical duration on a specific target platform. This permits the software engineers to communicate more effectively with the control engineers, since the properties of the system are closely aligned with the mathematical models used in control design.

The straightforwardness of the deployment of a LET-based model into a TTA-based platform comes from the fact that like the TTA, the LET MoC is based on a time-triggered formalism. Besides facilitating its mapping into TTA platforms, this property implies that there are a number of previously known points in time when fault-tolerance mechanisms are executed. This knowledge enables systems to directly discover errors such as message losses or duplications. The implementation of fault-tolerance mechanisms is therefore simplified in TTA-based platforms.

Another advantage of the LET MoC is that, by definition, it avoids the need for synchronization during the execution of jobs and triggers all the jobs synchronously. This reduces the number of failures that might occur in the system, since the absence of failures due to faulty synchronization points and different orders of execution is guaranteed by the MoC.

Architecture

The PIMs in PS-TTM rely on the architectural hierarchy described in [PNOES10a], where computational components can be defined as *systems*, *distributed application subsystems* (*DAS*es) and *jobs*.

- **Hierarchical components:**
  - ***Systems*** are closed h-components, i.e., they provide no interface for communication purposes with other components. Thus, platform independent models in PS-TTM usually contain a unique *System*, which gives the top-level view of the hierarchical system.
  - ***DASes*** are open h-components, i.e., they do provide the LIF, CP and DM interfaces in order to enable the communication with other components. Hence, PIMs in PS-TTM tipically contain more than one DAS, each of them providing a well-specified application service from the overall system functionality.
- **Atomic components:**
  - ***Jobs*** are atomic components that read the inputs, perform the desired computation and provide the corresponding outputs to the communication channels, following the LET MoC.

An example of a PIM architecture can be seen in Figure 4.5, where the distinction between different interfaces and the i-components of DASes are omitted for simplicity.

Execution and communication: The PI-TTM

In order to perform the simulation of PIMs that rely on the LET MoC in SystemC, the PS-TTM includes a novel LET based simulation engine called Platform Independent Time-Triggered Model (PI-TTM) [ANPP14]. The PI-TTM library handles the simulation of the LET-based models in SystemC by means of the E-TTM execution engine. The E-TTM execution engine applies the temporal restrictions imposed by the LET MoC to the models. In other words, the PI-TTM engine behaves as a layer above the E-TTM engine that abstracts the E-TTM details and provides a clear interface for the design of LET models to the engineers.

LET models are cyclic over time, i.e., their behavior can be described as a periodically repeating sequence of actions. Therefore, *System* components in PI-TTM contain

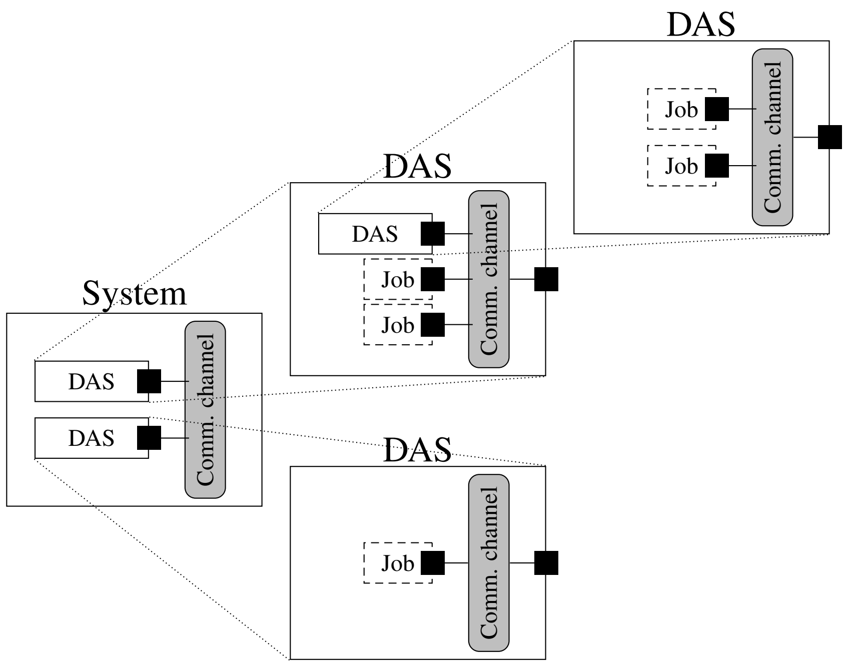


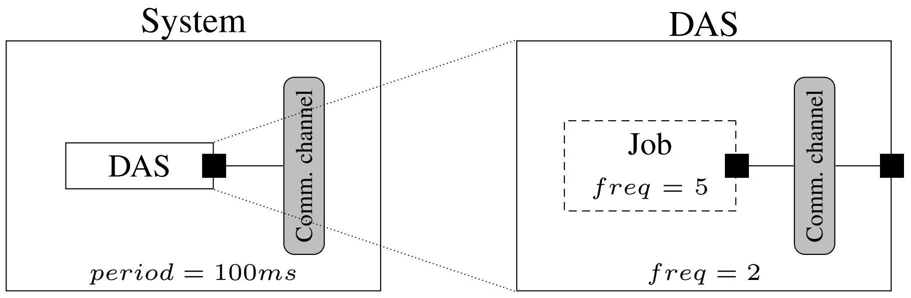
Figure 4.5: Example of a PIM in the PS-TTM

an attribute named *period*, which must be set by the designers and establishes the logical execution time of the system. *DASes* and *jobs*, however, specify their *frequency*, which states how many times the component is executed per period of its enclosing (sub)system. Therefore, the frequency of a job, together with the period of the hierarchical component where it is located, determine its LET, as equation 4.1 shows (where h_freq means hierarchically accumulated frequency).

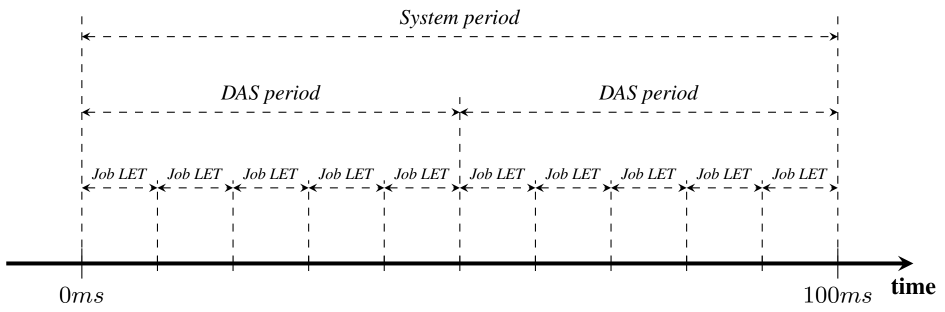
$$LET_{job_i} = \frac{period_{system}}{h\_freq_{job_i}} \tag{4.1}$$

Figure 4.6 shows an example of this. In the model of Figure 4.6a, a *System* with a period of *100ms* contains a *DAS* of $freq = 2$ which in turn contains a *job* with $freq = 5$. In that case, the job is executed 5 times in each DAS execution cycle, that is, $2 \cdot 5 = 10$ times per system period. Hence, the LET of the *job* is $10ms$, as Figure 4.6b shows.

When the models are defined according to the LET paradigm, computational components read their inputs instantaneously at the beginning of their execution, and they provide their outputs at the end of their LET. Hence, each computational job in LET systems introduces a delay to the final response time. This delay is equal to the logical execution time of the job. The sum of the logical execution times of all the jobs that are executed serially establishes the minimum period of the system.



(a) LET-based hierarchical PIM model



(b) Hierarchical Logical Execution Times

Figure 4.6: Logical Execution Times in PS-TTM

However, in order to support the design of hierarchical systems without leading to unacceptable delay times for real-time systems and without introducing undesired unambiguities for safety-critical systems to the models, hierarchical components in PI-TTM are defined as virtual components, i.e., they are used to facilitate a visual composition of the system but they do not introduce a delay on the simulated time of the execution. In other words, hierarchical components in PI-TTM are used for cognitive complexity management purposes, but they do not affect the functionality of the system, so this can be understood as a completely flat composition of different jobs.

The simulation of LET-based models in SystemC is therefore handled by the PI-TTM, which automatically manages the schedule of the LET-based platform independent model. To do so, the PI-TTM automatically creates and connects a new clock for each job during the initialization phase of the simulation. The clock is configured according to the logical execution time of the job, triggering a new clock event every logical every time the job has to be executed.

Communication between components is performed by pushing/pulling shared variables stored in the communication channels of hierarchical components. The temporal constraints imposed by the LET MoC are guaranteed by the automatic communication channel management of the PI-TTM execution engine, which delays push actions until one delta-cycle before the next triggering instant of the job, whereas it performs pull actions instantaneously one delta-cycle before the execution of the jobs. This way, the PI-TTM works as an intermediate layer between the model and the E-TTM engine, ensuring that the simulation adheres to the LET constraints of the application.

Synchronous jobs are executed sequentially making use of the delta-delay concept of SystemC. The partition between communication and computation provided by the PS-TTM framework and the automatic management of the communications provided by the PI-TTM ensures that the outputs provided by a given job are not made available for the rest of the jobs during that delta delay. This approach guarantees that all the simultaneous jobs are sequentially executed before any of their input values is updated, which implies that the scheduling order of the jobs does not affect the final result and therefore guarantees the time and value determinism of the model. Moreover, the PI-TTM greatly simplifies the design of LET models for the engineers, since the handling of the communications is automatically performed by the execution engine and it is not required to be done explicitly.

The PI-TTM also determines the maximum granularity of the simulation automatically, by dividing the period of the system by the least common multiple of the hierarchically accumulated frequencies of all the jobs within the system, as equations 4.2 and 4.3 show. This makes possible to perform the fastest possible simulations while preserving time and value determinism.

$$LCM_{system} = \forall\left(job_i, job_{i+1} \in system\right), LCM\left(h\_freq_{job_i}, h\_freq_{job_{i+1}}\right) \tag{4.2}$$

$$sim\_granularity = \frac{period_{system}}{LCM_{system}} \tag{4.3}$$

Figure 4.7 shows how LET-based models are managed by the PI-TTM library. In the example, two LET jobs with different frequencies communicate with each other. Job 1 has frequency 1 and job 2 has frequency 2. As mentioned, the PI-TTM automatically sets the trigger instants for each job and delays the update of each variable according to the LET of the jobs. In this case, it sets the granularity of the simulation to the half of the period, triggers job 1 every two macroticks and job 2 every macrotick, and sets the corresponding delays to their outgoing messages (1 and 0 respectively). Note that the dotted sections on the time axis represent zero advance of time, so the communication in LET models and the execution of jobs in E-TTM models are instantaneous.

Syntax

The PS-TTM meta-model has been built as a library that extends SystemC with a set of macros and functions that helps the designers in the definition of the models. Besides

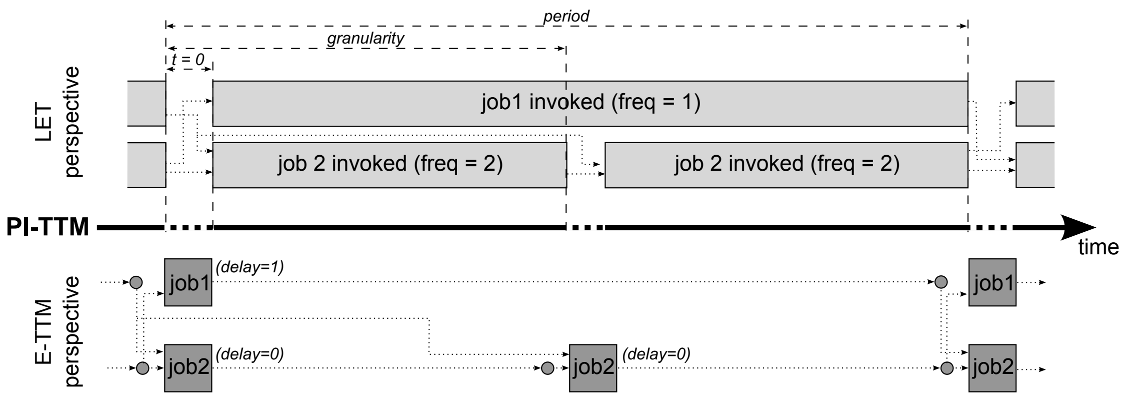


Figure 4.7: PI-TTM: Management of LET-based PIMs

this, it also contains the PI-TTM execution engine for LET-based models. In the following, we briefly introduce the interface and the functions provided by the PS-TTM environment to define and simulate LET-based platform independent models in SystemC with PI-TTM.

- **Systems:**
  - **Defining a system:**
    Systems are defined with the following macro:

    ```
    PITTM_SYSTEM(system_name)
    ```

    This macro requires to fill the internally defined `v_c` vector, which must contain the "child" components (DASes and jobs) of the hierarchical system.
  - **Instantiating a system:**
    A new constructor has to be defined for each *system*, which must be defined as a child function (by inheritance) of the default constructor for PIMs:

    ```
    pittm_system::pittm_system (
        const sc_module_name& snm,
        ttm_dt_time period);
    ```

    This constructor requires to set a name for the system and its period.
  - **Initializing a system:**
    The definition of the system requires, as a final step, to be initialized by calling the following initialization function:

    ```
    void pittm_system::initialize (
        const ttm_vector_rn& vector_rn_rte);
    ```

The vector *vector_rn_rte* must define the shared variables of the system, in order to enable the different sub-components of the system to communicate among themselves. This command is responsible of creating and configuring the clocks for each job and is also in charge of connecting the internal sub-components of the system together during the initialization phase of the simulation.

  - **Building a system:**
    The whole system is fully defined when the building command is called:

    ```
    void pittm_build_system (ttm_component * system);
    ```

    This function terminates the definition of the system. The initialization phase of the simulation is finished when the system is built. At that moment, the execution of the simulation can begin by calling the native SystemC `sc_start();` command.

- **DASes:**

  - **Defining a DAS:**
    The definition of DASes is carried out by using the following macro:

    ```
    PITTM_DAS(das_name)
    ```

    Similarly to the *systems* case, this macro requires to fill the `v_c` vector, which has to contain a reference to the components inside the DAS (other DASes or jobs).

  - **Instantiating a DAS:**
    Each DAS has to define at least one constructor, which must be declared as a child function of the default constructor for DASes:

    ```
    pittm_das::pittm_das (
        const sc_module_name& snm,
        unsigned int freq);
    ```

    The required parameters for this constructor are a name and a frequency for each DAS. It is not allowed to give the same name to more than one DAS placed in the same hierarchical component.

  - **Initializing a DAS:**
    The final step of the definition of a DAS must be its initialization. To do so, the initialization function is used:

    ```
    void pittm_das::initialize (
        const ttm_vector_rn& vector_rn_rte,
        const ttm_dt_com_route& route);
    ```

This initialization command is in charge of creating and configuring the clocks for each job, and of connecting all the internal sub-components of the system together during the initialization phase of the simulation. The *vector_rn_rte* must define the shared variables of the DAS, in order to enable communication among different sub-components. In addition, the *route* vector has to define the connections from/to the outside of the DAS to be managed by the interface component.

- **Jobs:**

  - **Defining a Job:**
    The definition of LET-based jobs in PI-TTM is enabled by their macro:

    ```
    PITTM_JOB(job_name)
    ```

    The main requirement of this macro is the definition of a clocked thread in the job, which is usually called `void ctask_thread(void);`. This thread hast to be defined as an infinite loop containing the periodic behavior of the job (read inputs, perform a computation, write outputs), and must contain a `wait()` command. This way, during the execution, the PI-TTM triggers the clocked thread of each job at the corresponding instant, which is executed instantaneously until a *wait()* command is found. This command determines the end of the cycle of the job and gives the control of the simulation back to the PI-TTM execution engine, which automatically triggers the next job according to the schedule. When all jobs that need to be triggered at a given instant have been processed, the PI-TTM advances its clock to the instant in which the next job has to be triggered, and forwards the push operations performed by the jobs starting at that instant.

  - **Instantiating a Job:**
    The constructors created for each job must inherit from the default job constructor:

    ```
    pittm_job::pittm_job (
        const sc_module_name& snm,
        unsigned int freq);
    ```

    Similarly to the *DAS* constructor, the default constructor for jobs requires to set a name and a frequency for each job instance. Jobs within the same h-component must have unique names.

  - **Communicating:** For communication purposes, the jobs can read/write data from/to the communication channels of the hierarchical component they are placed in.

* **Reading data:**
  The templated `pull` command is used for reading purposes. The command is formally defined as follows:
  ```
  template<typename T>
  T pull (
      e_ttm_com_if if_id,
      const ttm_dt_rn& rn_rte)
  ```
  Thus, the jobs can read the data from the communication channel they are connected to by specifying the interface to perform the reading (LIF, CP or DM) and the name of the shared variable to read.
* **Writing data:**
  The `push` command is the mechanism to write data in PI-TTM models. The command is defined as:
  ```
  template<typename T>
  void push (
      e_ttm_com_if if_id,
      const ttm_dt_rn& rn_rte,
      const T& value)
  ```
  Therefore, in order to write data, jobs have to specify the interface to which they want to write the data, the name of the shared real-time entity to overwrite, and the value.

Listing 4.1 shows an example of the definition of a platform independent model in PI-TTM. The example shows a system that will run at a period of `per`, and contains two DASes, the first of frequency 2 and the second of frequency 5. The first DAS is composed by two jobs with a frequency of 1 and 2 respectively. Job A reads two variables from the LIF interface, `DIST_A1` and `DIST_A2`, and calculates the average distance from these values in the `fcn(...)` function. Once calculated, it pushes back the value via its LIF interface and waits until the next time it is triggered.

Platform Specific Models

According to our modeling work-flow proposed in section 4.2, once the PIM of the system has been fully verified by means of the PI-TTM simulation engine, and the behavior against all the test cases and SFI campaigns is considered successful, the components of the platform independent model are deployed onto a model of the platform, thus creating the platform specific model of the system. For that reason, the PSM can be considered a refined version of the PIM, more than a different model of the system.

However, the development of a platform specific model with PS-TTM provides several benefits to the designers. First, the PSM enables them to describe the timing properties of the system at a more detailed level, which gives them the freedom to adjust

Listing 4.1: Example code of a PIM in PI-TTM

```
/* example_system.h */
PITTM_SYSTEM(Example_System){
    ...
    /* Constructor */
    Example_System(const sc_module_name & nm, ttm_dt_time period) : pittm_system(nm, per) {
        /* Declare Components */
        DAS_1   *p_das_1    = new DAS_1 ("das_1", 2);
        DAS_2   *p_das_2    = new DAS_2 ("das_2", 5);
        /* Place components in hierarchical component */
        v_c.push_back(p_das_1);
        v_c.push_back(p_das_2);
        ...
        //Initialize
        initialize(Example_System_Variables_rte, route);
    }
};

/* das_1.h */
PITTM_DAS(DAS_1){
    ...
    /* Constructor */
    DAS_1(const sc_module_name & nm, unsigned int freq) : pittm_das(nm, freq) {
        /* Declare Components */
        JOB_A   *p_job_a    = new JOB_A ("job_a_inst", 1);
        JOB_B   *p_job_b    = new JOB_B ("job_b_inst", 2);
        /* Place components in hierarchical component */
        v_c.push_back(p_job_a);
        v_c.push_back(p_job_b);
        ...
        //Initialize
        initialize(DAS_1_Variables_rte, route);
    }
};
/*!< Real-time entities of the DAS*/
ttm_dt_rn * DIST_A1 = new ttm_dt_rn("DIST_A1");
ttm_dt_rn * DIST_A2 = new ttm_dt_rn("DIST_A2");
ttm_dt_rn * DIST_AV = new ttm_dt_rn("DIST_AV");

/* job_a.h */
PITTM_JOB(JOB_A){
    /* Internal variables */
    tDistanceInt32 dist_1, dist_2;
    tDistanceInt32 av_dist;
    /* Constructor */
    JOB_A(const sc_module_name & snm, unsigned int freq) : pittm_job(snm, freq){}
    /* Thread */
protected:
    void ctask_thread(void);
    void fcn(void);
};

/* job_a.cpp */
void JOB_A::fcn(){ av_dist = (dist_1 + dist_2) / 2; }
void JOB_A::ctask_thread() {
    ...
    while(true) {
        /* pull variables */
        dist_1 = pull<tDistanceInt32>(e_TTM_COM_IF_LIF, DIST_A1);
        dist_2 = pull<tDistanceInt32>(e_TTM_COM_IF_LIF, DIST_A2);
        /* compute */
        fcn();
        /* push variables */
        push<kcg_bool>(e_TTM_COM_IF_LIF, DIST_AV, av_dist);
        /* wait LET */
        wait();
    }
}
```

the response of the system to their needs more accurately. Second, mapping a given PIM into different platform candidates and simulating each resulting PSM facilitates to perform a comparison of the behavior of each model in order to take a decision about the most adequate system for their specific needs. Third, the PSM specifies how the functional model makes use of the chosen platform, which enables the testing teams to apply different HW-related phenomena that affect the behavior of the system to the simulation, such as jitter or faults in HW components.

The following sections describe the characteristics of platform specific models and their simulation in PS-TTM in detail.

Underlying Model of Computation

In order to provide a more accurate control of the timing specifications of the different components of the system to the designers, the platform specific models in PS-TTM are designed according to the time-triggered MoC introduced in [EBK03], which relies on the sparse model of time.

Consequently, the flow of time of PSMs is divided into alternating intervals of activity and silence. As mentioned in section 2.5, in order to guarantee that a model designed according to the TT MoC reasonably represents the final system and the system behaves deterministically, the HW platform must fulfill a number of prerequisites. Since TTA-based platforms comply with such conditions, relying on the TT MoC guarantees the straightforwardness of the deployment of the models into TTA-based platforms.

Besides, although the TT MoC-based models might be slightly more complex than the LET-based models from a cognitive point of view, the complexity management techniques proposed by the TT paradigm enable to keep the complexity of the platform specific models at a reasonable level.

Moreover, the fact that the LET-based jobs designed in the PIM are treated by the PI-TTM engine as time-triggered jobs to which the temporal restrictions imposed by the LET MoC have been applied, simplifies their transformation to TT MoC based jobs. Therefore, the TT MoC fits well as an intermediate MoC between the preliminary LET-based PIMs and the final system.

Therefore, all these characteristics make the TT MoC an ideal candidate for the design of Real-Time Safety-Critical systems at a platform specific level.

For these reasons, this approach, together with the work-flow presented in section 4.2, facilitate the modeling and validation of time-triggered systems. It starts with the design of a purely functional LET-based model and enables a straightforward transformation into a platform specific TT MoC-based model, what guarantees that time-properties are intrinsically preserved in the final implementation when the platform is based on the TTA.

Architecture

Analogously to the case of the PIMs, platform specific models are defined hierarchically in PS-TTM. The meta-model of the PS-TTM subdivides the computational components of the PSMs into the following classes:

- Hierarchical components:
  - ***Systems***: *Systems* are closed h-components, i.e., they provide no interface to communicate with other components. Hence, platform specific models in PS-TTM are designed as a *System*, which gives the top-level view of the hierarchical system. *System* components represent the complete real-time time-triggered system.
  - ***Clusters***: *Clusters* are modeled as open h-components that provide the LIF, CP and DM interfaces in order to enable the communication with other components. TTA systems may be composed by one or several different clusters.
  - ***Nodes***: *Nodes* are the basic building blocks of TTA-based systems. Clusters of TTA-based systems are typically decomposed in different nodes interconnected by a time-triggered communication channel (Figure 2.12), each of them performing a well-defined task.
  - ***Processors***: *Nodes* of time-triggered systems are composed by a Communication Controller (CC), Communication Network Interface (CNI) and a host processor that executes the application software. Therefore, *processor* components of the PS-TTM represent the computational processing units (CPUs) of the nodes, and they are designed as open hierarchical components that provide the necessary interfaces for communication with other components (LIF), configuration (CP) and maintenance (DM).
  - ***Cores***: Traditionally, CPUs contained a single core performing the computations specified by the software. However, the need of a higher computational power and speed resulted in the commercialization of multi-core chips in the mid noughties. Nowadays, even many-core chips are being launched to the market in order to increase the speed of systems. Thus, *core* components of PS-TTM, modeled as open-hierarchical components, enable designers to design the processors of their systems modularly.
  - ***Hypervisors***: Hypervisors are software programs that virtualize the architecture they run on, and enable to define different partitions where software can be mapped to. Due to the configurability they offer and the spatial isolation they provide, hypervisors are a promising approach for the design of safety-critical embedded systems, as different research projects [CRM+09, MRCP10, Van10] and tools [wwwm] demonstrate. *Hypervisors*

are defined as open h-components in PS-TTM too, and are composed of one or more *partitions*.

  - ***Partitions***: Partitions are virtualized HW parts defined and managed by hypervisors. Therefore, the PS-TTM models them as open hierarchical components that provide LIF, CP and DM interfaces. The definition of a *hypervisor* in a PS-TTM models also requires the designers to define at least one *partition* inside it.

- Atomic components:

  - ***Jobs***: *Jobs* are the unique atomic components of platform specific models in PS-TTM, and must be deployed in *cores* or *partitions* of the model. Analogously to platform independent *jobs*, *jobs* of PSMs read their inputs, perform the computation defined by their software task and provide the corresponding outputs to the communication channels. Following the time-triggered MoC, the execution of jobs takes place during the activity interval of the sparse time.

HW components of a TTA system are represented by h-components of a PSM. These h-components are defined as SystemC wrappers, providing an high-level abstraction of the HW components. However, the developers are enabled to extend their definition in order to get a more representative description of their specific component model.

The target-platform models (PMs) are designed by assembling a hierarchical composition of these HW-related components. Once the PMs are built, the designers can construct the PSMs by deploying the jobs designed in the PIM into the components of the PM.

Figure 4.8 presents an example of the modeling of a hierarchical PSM in PS-TTM. As the figure shows, the development of the platform starts with the definition of a *system*. A number of *clusters*, *nodes*, *processors*, *cores* can be included in the system at each hierarchical level. Optionally, *cores* may contain a *hypervisor* with some *partitions*. The PSM is built when the jobs that define the functionality of the system are mapped into the *cores* or the *partitions* of the system. The different interfaces (LIF, CP and DM) and the i-components of the h-components are omitted from the figure for clarity.

Execution and communication

In order to simulate TT MoC-based platform specific models, a novel simulation engine has been designed, called Platform Specific Time-Triggered Model (PS-TTM) engine. Compared to the PI-TTM execution engine used in PIMs, the PS-TTM simulator gives a more accurate control of the timing properties to the designers, which are enabled to define temporal phenomena such as phase delays or jitters in components.

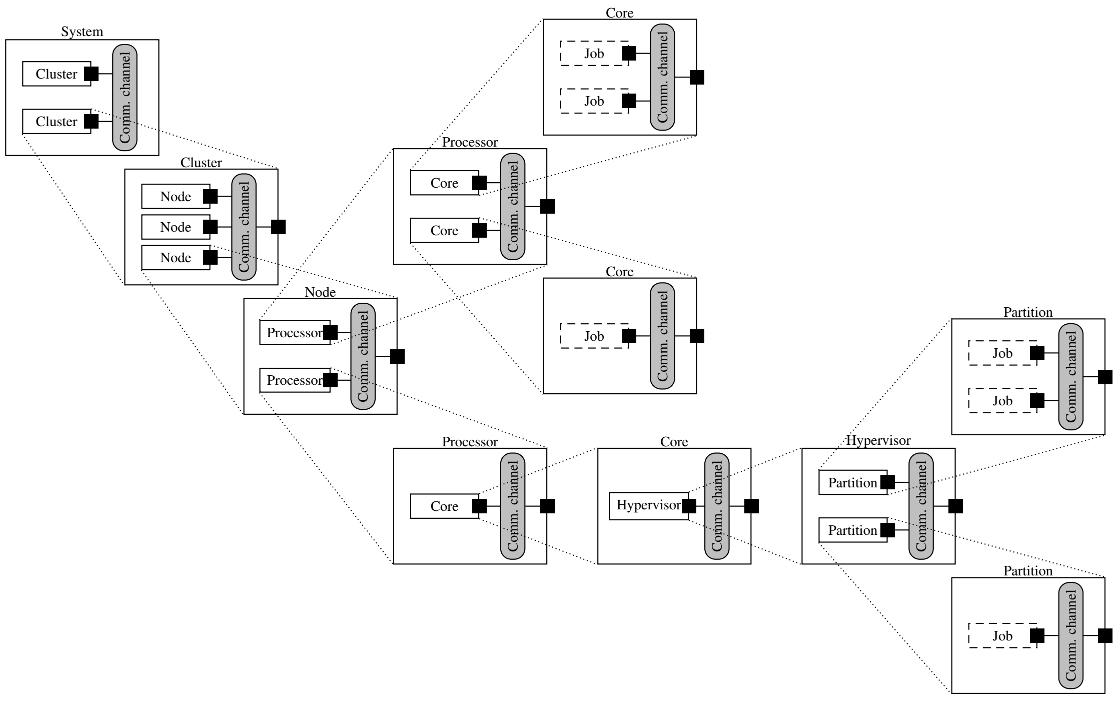


Figure 4.8: Example of a PSM in the PS-TTM

As mentioned previously, the PSMs rely on the time-triggered MoC in PS-TTM. Consequently, the PS-TTM engine provides a strict separation between communication and computation and establishes a global notion of time, based on the sparse-time concept, which divides the flow of time in PSMs in alternating intervals of activity and silence.

The meta-model of PS-TTM is developed as a C++ library that extends the E-TTM environment with abstract and extendable models of platform components. The usage of this library of HW components in platform specific models gets the biggest relevance when assessing the reliability achieved by the system due to its fault-tolerance mechanisms. The non-intrusive simulated fault injection technique provided by the PS-TTM ATE (see Chapter 5) enables testing teams to validate the appropriateness of the fault-tolerance mechanisms introduced in the system against different effects of faults in HW components.

Simulation is carried out by means of the PS-TTM simulation engine, which strongly relies on the E-TTM [PPO10]. The sparse-time-based global notion of time is based on the global time provided by SystemC. Thus, the simulation may be faster, slower or at the same pace as physical time (given by the operating system of the host computer running the simulation).

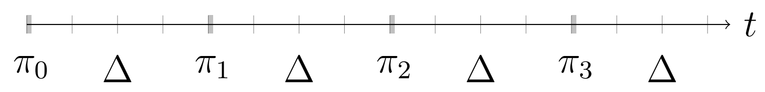


Figure 4.9: Sparse time model in E-TTM ($\pi = 0, \Delta = 4$)

According to the E-TTM framework, the execution of computational components is restricted to the activity intervals (π) of the sparse-time abstraction model, and it is forbidden to execute them between macroticks, i.e., during the silence interval (Δ). Internal events are also restricted to the activity interval. In case an external event (such as an event from an external model relying on a different MoC) occurs outside that interval, its activation is delayed until next activity interval.

The activity interval π has a duration of zero simulation time (Figure 4.9). This means that jobs, which are restricted to take place during the activity interval, are executed instantaneously. Although simultaneously triggered components are executed sequentially, this sequential execution is performed in zero simulation time. Besides, the output messages are not delivered during the execution macrotick and the events generated during the computation do not trigger activation events until next execution macrotick. Therefore, race conditions are avoided by definition, which guarantees that the execution order chosen by the E-TTM scheduler for simultaneously triggered jobs does not have any impact on the outputs of the system. Thus, it is guaranteed that time determinism will not be violated, regardless of the scheduling order chosen by the simulation engine.

The execution of jobs is configured by the designers, by specifying a *period* for each of them. The *period* establishes the duration of each job. When, following the work-flow specified in section 4.2, the developers design their PSM from the PIM, the calculation of the period corresponding to a job is facilitated by its LET, which can be obtained by equation 4.4.

$$period_{job_i} = LET_{job_i} = \frac{period_{system}}{h\_freq_{job_i}} \tag{4.4}$$

Besides, jobs at PSM level must specify a relative *phase*, which establishes the *phase* difference between the start of the cycle and the effective instant when the job is executed. The *phase* delay is specified as a percentage value ($0.0 \leq phase \leq 100.0$), where a value of 0.0 establishes that the execution of the job will take place at the beginning of the cycle and a value of 100.0 sets the execution at a delta delay before the end of the cycle.

When deriving the PSM from the PIM, the *phase* of all jobs is usually set to 0.0, in order to obtain the equivalent timing properties in the PSM. The PS-TTM gives the designers the freedom to modify the value of the *phase* of each job as desired. This specification is made in the initialization function of the hierarchical component containing the job. Thus, different instances of a given job can have different phase delays. Setting a $phase$ to a value different from 0.0 might be interesting in the following cases:

- In case a job of the PIM is manually split into more jobs in the PSM, the *phase* delay of the PSM jobs may be modified in order to respect the causality chain of the original PIM in the PSM model consisting of a set of jobs executing sequentially.
- To manually set the scheduling order of jobs.
- To simulate different phenomena such as imperfect synchronization between the different clocks of the system.

However, the developers must bear in mind that modifying the *phases* may have a big impact and entirely modify the behavior of the system. Therefore, the modification of the *phase* values of the system is not recommended, and, if done, the emerging behavior of the platform specific model must be carefully validated again.

Communication between components is again performed by *push/pull* operations in shared variables stored in the communication channels, via the LIF, CP and DM interfaces. Whenever a component pushes a new value (during activity interval), the value is instantaneously given to the communication infrastructure.

The simulation engine requires the designers to specify a delay in the transmission of information between components. This delay represents the amount of time between the instant in which a given job starts its execution and the instant in which its output values are available for the rest of the components in the communication channels. The delay is specified in number of macroticks of the simulation. When, as suggested by the work-flow specified in section 4.2, the developers design their PSM from the PIM, the calculation of the appropriate delay for each push activity becomes straightforward, and can be obtained by dividing the Least Common Multiple (LCM) among all the total frequencies of the jobs within the system by the total frequency of the job executing the push command, as equation 4.5 shows.

$$delay_{job_i} = \frac{LCM_{system}}{h\_freq_{job_i}} \tag{4.5}$$

And if we consider equations 4.3 and 4.4, the delay time for the communications of each job can also be calculated by their period and the granularity of the simulation, as shown by equation 4.6.

$$delay_{job_i} = \frac{period_{job_i}}{sim\_granularity} \tag{4.6}$$

However, in case a *job* is not directly derived from the platform independent model and does not specify a *phase* equal to 0.0, the appropriate communication delay should be calculated by the designers.

Syntax

The PS-TTM meta-model provides a set of macros and functions on top of SystemC for the development of the platform specific models. We briefly introduce the most important functions and commands for the definition of time-triggered MoC based PSMs provided by the PS-TTM framework below:

- **Systems**:
  - **Defining a system**:
    Systems are defined by the following macro:

    ```
    PSTTM_SYSTEM(system_name);
    ```

    Similarly to the case of PIM systems, this macro requires to fill the internal `v_c` vector specifying the clusters that are inside the system.
  - **Instantiating a system**:
    Systems must declare a new constructor, which has to be defined as a child function of the default constructor for PSMs by inheritance. The default constructor only requires to set a name for the system:

    ```
    psttm_system::psttm_system (
        const sc_module_name& snm );
    ```

    In contrast to PIM systems, the instantiation of PSM systems does not require to define their period. This is because whereas the period (or LET) of jobs in PIM systems is automatically calculated by their frequency respect to the period of the hierarchical component they are place in, the period (and phase) of PSM jobs is explicitly specified when they are called by a h-component (see section "Initializing an open h-component" below). Thus, no period is specified for PSM systems, since jobs themselves specify their own period.
  - **Initializing a system**:
    Systems must include the initialization function at the end of their definition, which is defined as follows:

    ```
    void psttm_system::initialize (
    const ttm_vector_rn& vector_rn_rte );
    ```

    The initialization function requires to set the `vector_rn_rte` vector, which specifies the shared real-time entities of the system, in order to enable the communication between the internal components of the system via the communication channel.

- **Building a system**:
  The PS-TTM meta-model provides the following function to build the system:

  ```
  void psttm_build_system(ttm_component * system);
  ```

  This function terminates the definition of the system. The initialization phase of the simulation is finished when the system is built. Once the system has been built, the simulation can start by calling the native SystemC `sc_start();` command.

- **Open h-components** (Clusters, Nodes, Processors, Cores, Hypervisors & Partitions):

  - **Defining an open h-component**:
    The definition of open h-components is made by using corresponding macro for each component, namely:

    ```
    PSTTM_CLUSTER(cluster_name);
    PSTTM_NODE(node_name);
    PSTTM_PROCESSOR(proc_name);
    PSTTM_CORE(core_name);
    PSTTM_HYPERVISOR(hyp_name):
    PSTTM_PARTITION(part_name);
    ```

    Analogously to the *Systems* case, all these macros require the definition of `v_c` vector, which must specify their child components.

  - **Instantiating an open h-component**:
    Each open h-component must define its own constructor, which has to be declared as a derived function of its corresponding base constructor, among:

    ```
    psttm_cluster::psttm_cluster (
        const sc_module_name& snm);
    psttm_node::psttm_node (
        const sc_module_name& snm);
    psttm_processor::psttm_processor (
        const sc_module_name& snm);
    psttm_core::psttm_core (
        const sc_module_name& snm);
    psttm_hypervisor::psttm_hypervisor (
        const sc_module_name& snm);
    psttm_partition::psttm_partition (
        const sc_module_name& snm);
    ```

These default constructors only require to pass them the name of the component as a parameter. The designers are not allowed to repeat the names of different components in the same hierarchical system.

– **Initializing an open h-component**:
  Open h-components also have to include the initialization function as the final step of their definition. The initialization function is similar for *clusters*, *nodes*, *processors* and *hypervisors*, but differs for *cores* and *partitions*, as the following list shows:

```
void psttm_cluster::initialize (
    const ttm_vector_rn& vector_rn_rte,
    const ttm_dt_com_route& route);
void psttm_node::initialize (
    const ttm_vector_rn& vector_rn_rte,
    const ttm_dt_com_route& route);
void psttm_processor::initialize (
    const ttm_vector_rn& vector_rn_rte,
    const ttm_dt_com_route& route);
void psttm_core::initialize (
    const ttm_dt_time_cycle& stime_cycle_descr,
    const ttm_vector_rn& vector_rn_rte,
    const ttm_dt_com_route& route);
void psttm_hypervisor::initialize (
    const ttm_vector_rn& vector_rn_rte,
    const ttm_dt_com_route& route);
void psttm_partition::initialize (
    const ttm_dt_time_cycle& stime_cycle_descr,
    const ttm_vector_rn& vector_rn_rte,
    const ttm_dt_com_route& route);
```

  All the initialization functions have to specify their `vector_rn_rte` and `route` vectors. `vector_rn_rte` sets the shared real-time entities of the h-component, in order to enable the internal communication between the components within the hierarchical component, via their shared communication channel. The `route` vector must declare the connections from/to the outside of the open h-component, in order to establish a coherent communication between the open h-component and its outside.

  Besides, *clusters* and *partitions* must define their `stime_cycle_descr`, which is a vector that includes the temporal specifications of the jobs mapped into them. The `stime_cycle_descr` vector must include a period and a phase delay value for each of the jobs of the h-component.

- **Jobs**:
  - **Defining a Job**:
    Jobs of platform specific models are defined by the following macro:

    ```
    PSTTM_JOB(job_name)
    ```

    Inside the definition of the job, a clocked thread has to be defined, which is usually called `void ctask_thread(void);`. This thread hast to be defined as an infinite loop containing the periodic behavior of the job (typically: read inputs, perform a computation, write outputs), and must contain a `wait()` command. This way, the PS-TTM simulation engine triggers the clocked thread of each job at the corresponding instant, which is executed instantaneously till the *wait()* command. The command gives the control of the simulation back to the PS-TTM engine, which starts the execution of the next job according to the schedule.
  - **Instantiating a Job**:
    Each job has to define at least one constructor, which must be a derivative of the generic constructor for jobs at PSMs:

    ```
    psttm_job::psttm_job(const sc_module_name& snm)
    ```

    The default constructor for jobs requires to give a name to the job. Jobs placed inside a given hierarchical component must have unique names.
  - **Communicating**:
    Jobs can read/write data from/to the communication channel of their h-component at any point of their `void ctask_thread(void);` function. The communication channel is responsible of updating the values of the real-time entities when needed.
    * **Reading data**:
      The `pull` command is used to get a real-time image from the communication channel. This command is defined as follows:

      ```
      template<typename T>
      T pull(
          e_ttm_com_if if_id,
          const ttm_dt_rn& rn_rte)
      ```

      Jobs can read the data from the communication channel by specifying the interface to perform the reading (LIF, CP or DM) and the name of the shared real-time entity to read. The data is transferred to the job instantaneously (in zero simulation time).
    * **Writing data**:
      The `push` command enables the jobs to write data into the communication channels they are connected to. The command is defined as:

```
template<typename T>
void push(
    e_ttm_com_if if_id,
    const ttm_dt_rn& rn_rte,
    const T& value,
    double delay)
```

Thus, the mechanism to write data into the communication channel requires to specify the interface (LIF, CP or DM), the name of the real-time entity they want to overwrite, and the value the want to give to the variable. Besides, they must specify the updating delay, that is, the amount of time the communication channel should wait until updating the value of the real-time entity. In case the PSM has been directly derived from the PIM, i.e., no modification has been made to the timing properties of jobs (period and phase), the delay time can be calculated by equations 4.5 or 4.6. Otherwise, it would be responsibility of the designers to ensure the coherence of their models.

4.6 Mixed Abstraction-Level Simulation

Safety-critical systems are often big systems that are developed by different teams of HW and SW designers. Typically, each group of developers takes care of a part of the system, which is considered a subsystem, and takes the responsibility of modeling it. Once all the subsystems have been designed, they are assembled in a unique model that represents the whole system.

Therefore, it is usual to have the case in which different parts of the system evolve at a different pace, depending on their complexity and the amount of human and economical resources dedicated to their modeling. This fact might hinder the verification of the functional and non-functional properties of the subsystems, since the different levels of abstraction that the subsystem models might be at a certain moment hinders or even impedes their integration.

The PS-TTM provides a mechanism to get over this difficulty. In fact, as previously stated, the PI-TTM simulation engine has been built as an abstraction of the PS-TTM engine that transforms its model of computation into the Logical Execution Time. Thus, although these engines provide a different MoC simulator each, the fact that one is an abstraction of the other provides a seamless integration between them.

The PS-TTM makes use of this seamless integration between the two simulation engines to enable the designers to straightforwardly connect platform independent and platform specific models. This enables them to simulate the complete system even combining descriptions of subsystems at PIM and PSM design stages, which rely on different MoCs, and thus allowing to validate the functional and non-functional proper-

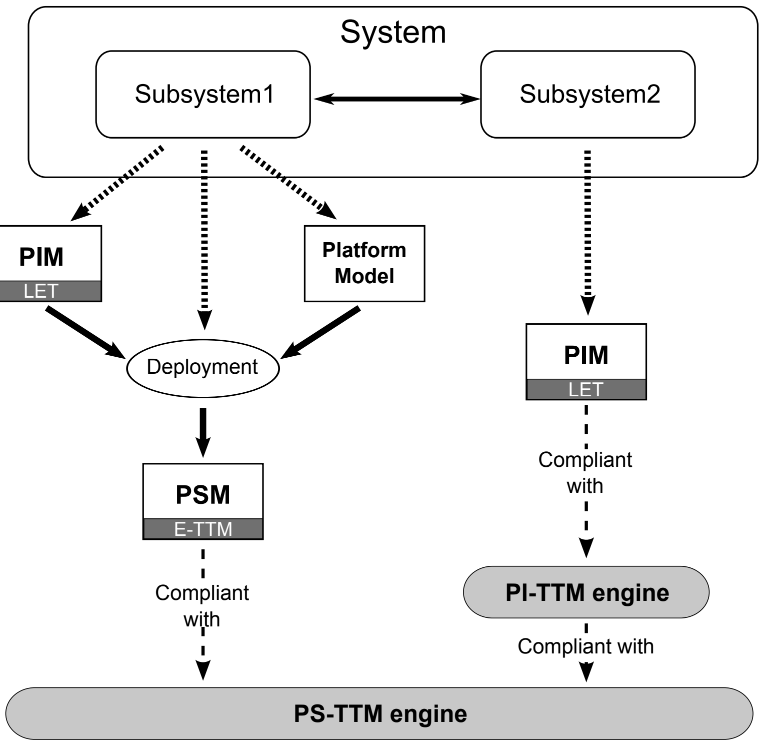


Figure 4.10: Mixed abstraction-level simulation with PS-TTM

ties of a single part of the system in a straightforward way. This capability is known as Mixed Abstraction-Level Simulation.

The mixed abstraction-level simulation is carried out by the PS-TTM simulation engine. The PS-TTM drives an additional PI-TTM engine, which enables to execute LET based PIM components. This way, PSM components based on the TT MoC run directly over the PS-TTM engine, whereas PIM components are managed by the PI-TTM engine running over the PS-TTM. In other words, the PS-TTM includes a PI-TTM engine that behaves a layer that translates LET based components into equivalent time-triggered components that the PS-TTM can run, as Figure 4.7 illustrated. The transformation of LET based components to TT based components follows the rules established by equations 4.4 and 4.6. The PI-TTM automatically sets the *phase* of all the PIM jobs to 0. Figure 4.10 graphically shows the architecture of the mixed abstraction simulation technique in the PS-TTM.

The figure shows how the PI-TTM library handles the simulation of the LET-based models in SystemC by means of the PS-TTM execution engine, applying the temporal restrictions imposed by the LET MoC to the models. As the figure shows, the PI-TTM engine behaves as a layer above the PS-TTM engine that abstracts the PS-TTM details to the designers, and performs the required transformations to the signals during simulation time in order to enable mixed abstraction-level simulation.

CHAPTER 5

Testing and Simulated Fault Injection Framework

The PS-TTM simulation platform includes a time-triggered testing and simulated fault injection framework, with the aim to provide the testing teams an environment to assess the fault-tolerance mechanisms implemented in their PS-TTM models. The testing and simulated fault injection tool is called PS-TTM Automatic Test Executor (ATE). This chapter describes the properties and specifications of the PS-TTM ATE.

5.1 The PS-TTM Automatic Test Executor (ATE)

The PS-TTM ATE has been built as an extension to the PS-TTM, with the goal of enabling an accurate evaluation of the functional and non-functional properties of the models in PS-TTM, with a special focus on their fault-tolerance properties.

The PS-TTM ATE gives the following three main properties to the testing teams:

- **Non-intrusiveness**: Non-intrusive fault injection techniques are the ones that completely mask their presence, so that they have no effect on the system behavior apart from the faults they inject. Therefore, the model is not modified to get the faults injected, which gives more reliable testing results.
- **Availability for PIM and PSMs**: Testing and simulated fault injection can be applied to both platform independent and platform specific models, in order to facilitate the detection of design flaws at the earliest stages of the design.
- **Repeatability**: The PS-TTM ATE provides repeatability to the testing and SFI activities, which enables to prove that the bugs found in previous versions of the models have been fixed in the newest versions.

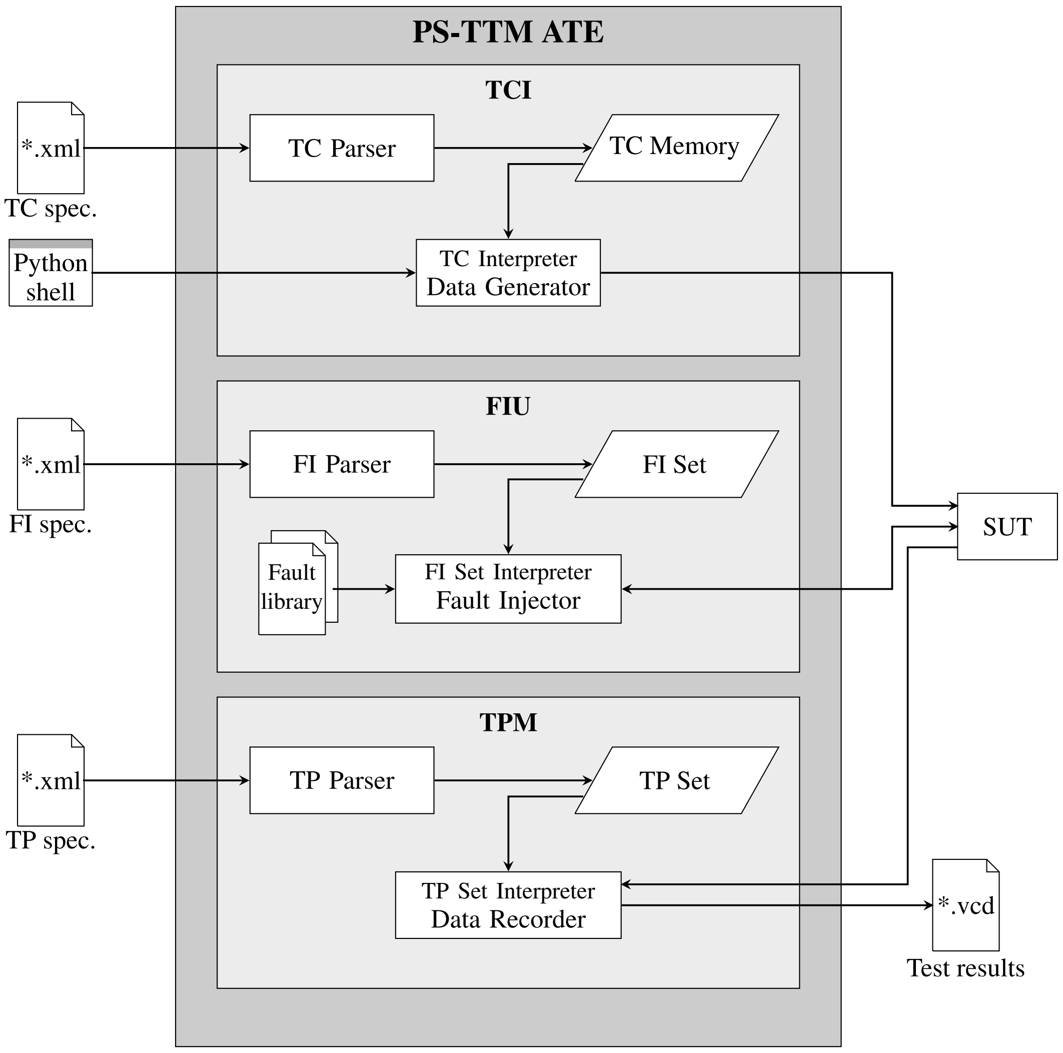


Figure 5.1: PS-TTM Automatic Test Executor

The PS-TTM ATE, which is shown in Figure 5.1, is composed by three different modules: The Test Case Interpreter (TCI), the Test Point Manager (TPM) and the Fault Injection Unit (FIU).

The interaction between the PS-TTM Automatic Test Executor and the test-engineers is performed by the interactive Python shell provided by the ATE. Each of the modules of the PS-TTM ATE provides different commands for the users to execute their functionality.

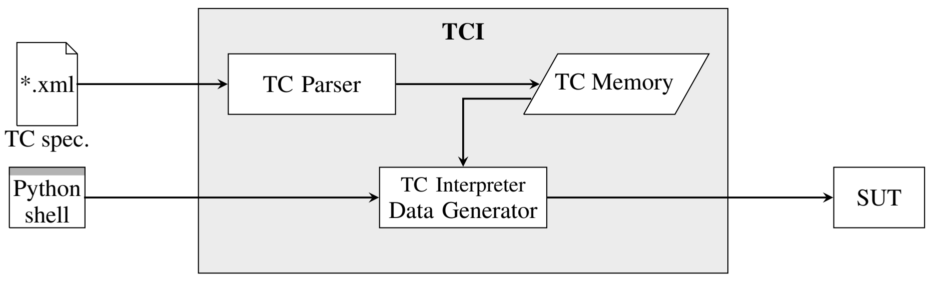


Figure 5.2: Test Case Interpreter Module

Test Case Interpreter (TCI)

The Test Case Interpreter is the module responsible for exercising the desired test cases. As figure 5.2 shows, the TCI is composed by three main components:

- Test Case Parser
- Test Case Memory
- Test Case Data Generator

The TCI is capable of automatically running different test cases defined by the testing engineers. During the initialization phase of the simulation, the test case parser of the TCI reads the test case specified by the test-engineers and stores it in the test case memory. Once the simulation starts, the data-generator feeds the SUT at each time tick with the signals corresponding to the test case stored in the test case memory.

Test cases are defined in XML files. The syntax of the test cases is simple: any time the test-engineers want to define a new value for an input, they have to first set the instant at which the new value should be available as an input for the SUT. For each time instant, any number of different inputs can be set. In case no value is set for an input at a given instant, the TCI feeds the SUT with the previous value of the corresponding input. Listing 5.1 shows an example of the syntax of these files.

User Interface

The Test Case Interpreter provides the following commands to interact with the ATE via the Python shell:

- `void LoadTestCaseDescriptionFile(string file)`: This instruction enables to specify which test case needs to be run by the TCI. The command accepts both absolute and relative paths to the test case configuration file, or, in case the desired test case configuration file is located in the same directory as the compiled PS-TTM model (executable file), just the name of the file.

Listing 5.1: Example of test case configuration file (XML file)

```
<!-- TEST CASE -->
<TestCase>
    <Setup instant="0.000">
        <Set Variable="Encoder1" Value="0"/>
        <Set Variable="Encoder2" Value="0"/>
        <Set Variable="Acceleration" Value="0"/>
        <Set Variable="NewBalise" Value="0"/>
        <Set Variable="BalisePosition" Value="0"/>
        <Set Variable="GroundInclination" Value="0"/>
        <Set Variable="NextBalisePosition" Value="0"/>
    </Setup>
    <!-- More inputs might be defined between these time stamps -->
    <Setup instant="95.750">
        <Set Variable="Encoder1" Value="361"/>
        <Set Variable="Encoder2" Value="361"/>
        <Set Variable="Acceleration" Value="1.97327"/>
    </Setup>
    <Setup instant="96.000">
        <Set Variable="Encoder1" Value="365"/>
        <Set Variable="Encoder2" Value="365"/>
        <Set Variable="Acceleration" Value="1.95508"/>
        <Set Variable="NewBalise" Value="1"/>
        <Set Variable="BalisePosition" Value="2000"/>
        <Set Variable="GroundInclination" Value="0"/>
        <Set Variable="NextBalisePosition" Value="3500"/>
    </Setup>
    <Setup instant="96.250">
        <Set Variable="Encoder1" Value="369"/>
        <Set Variable="Encoder2" Value="369"/>
        <Set Variable="Acceleration" Value="1.93706"/>
        <Set Variable="NewBalise" Value="0"/>
        <Set Variable="BalisePosition" Value="0"/>
        <Set Variable="GroundInclination" Value="0"/>
        <Set Variable="NextBalisePosition" Value="0"/>
    </Setup>
    <Setup instant="96.500">
        <Set Variable="Encoder1" Value="373"/>
        <Set Variable="Encoder2" Value="373"/>
        <Set Variable="Acceleration" Value="1.91921"/>
    </Setup>
    <Setup instant="96.750">
        <Set Variable="Encoder1" Value="377"/>
        <Set Variable="Encoder2" Value="377"/>
        <Set Variable="Acceleration" Value="1.90151"/>
    </Setup>
    <!-- Test Case continues... -->
</TestCase>
```

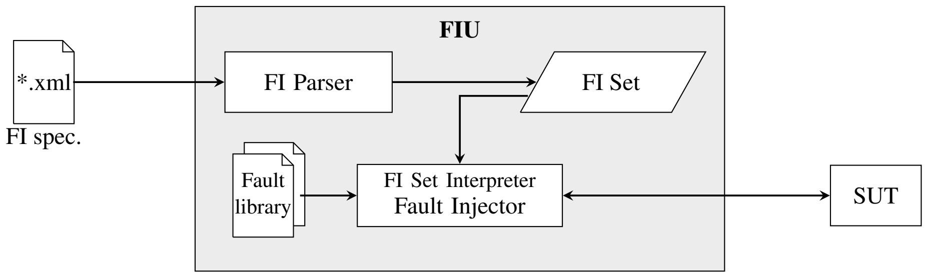


Figure 5.3: Fault Injection Unit Module

- `void Confirm(void)`: This command is used to notify the TCI that the setup of the test case configuration is finished. When this command is sent to the TCI, it informs the ATE that it is ready to start. When the ATE receives the same notification from the other modules, the initialization phase finishes and the simulation begins. Therefore, it is mandatory to send this command in order to ensure the simulation will start.
- `void SetSignal(string signal, float value)`: Besides the automatic interpretation of the test cases defined by test engineers, the TCI enables setting the input signals of the SUT interactively during simulation. This command is used for that purpose. In case a non-existing signal is specified by the user, the command is ignored.

Fault Injection Unit (FIU)

The FIU provides simulated fault injection capabilities to the ATE. Specifically, the FIU enables the test-engineers injecting different types of faults in the internal signals of the SUT. As figure 5.3 shows, this module is composed by 4 components:

- Fault Injection Parser
- Fault Injection Set
- FI Set Interpreter & Fault Injector
- Fault Libraries

The fault injection parser reads the fault configuration files (XML files) during the initialization phase of the simulation, and stores the fault injection campaign defined by the test engineers in its memory. Once the simulation starts, the PS-TTM engine automatically diverts the signals inside the SUT to the FI Set Interpreter & Fault Injector, which compares the properties of the received signals and the current time-stamp of the simulation with the Fault Injection Set stored in the memory. In case a fault has to be injected, the fault injector sabotages the signal as required according to the fault effect specified in the fault injection campaign. To do so, the FIU includes a set of fault libraries for both platform independent and platform specific models. Once the signal has been corrupted, it is sent back to the SUT. This way, the fault injection process follows a non-intrusive approach, since the model of the system does not suffer any modification for the fault injection activities.

Besides, this process is performed in zero simulation time, which guarantees that the temporal properties of the system are not affected by the fault injection process and remain intact when faults are applied.

The FIU supports two different fault modes: transient and permanent. Permanent faults are assumed to remain active until the simulation ends, and therefore they only need to specify their trigger time. However, Transient faults are temporary misbehaviors, so their configuration requires to specify a duration in addition the triggering instant. Once the injection of a transient fault is finished, the signal affected by the fault returns back to a non-faulty state.

The selected XML schema for the definition of fault injection campaigns complies with the international ASAM AE HIL standard for hardware-in-the-loop testing. Although the aim of this work is not to perform fault injection at hardware-in-the-loop level, sticking to the standard enables forward reuse of the fault injection campaigns until the final prototyping phase. In order to ease the definition of the fault injection specification, a graphical user interface and automatic XML code generation tool has been developed in this work, which is further described in chapter 6. Anyway, since the FI campaigns can be defined by hand, the following paragraphs describe the schema followed by the fault-configuration files. Listing 5.2 shows an example of a fault configuration file.

The fault injection campaigns are structured in three blocks: A fault configuration with fault sets, faults and locations.

The fault configuration may include one or more fault sets. Each fault set, defined by the *FaultSet* tag, specifies a *name*, a *fault mode*, a *triggering instant*, and references to one or more faults. The fault mode must be specified either as `TRANSIENT` or `PERMANENT`. Therefore, in case the *fault mode* is set to `TRANSIENT` the fault set must also specify a *duration* for the fault in addition to the rest of configuration parameters.

A set of faults might be defined in the *Faults* block. Each fault must specify a unique *id*, a *name*, a *fault effect* from the fault libraries (see section 5.2) and a set of attributes that depend on the specific effect of the fault. In case the fault configuration is defined by means of the GUI, the XML generator automatically sets the unique *id* of each fault in order to avoid duplicates.

The last block specifies a set of locations where the faults might be injected. Each location must define a unique *id*, which is also given by the GUI, and the hierarchical

Listing 5.2: Example of fault configuration file (XML file)

```
<!-- FAULT CONFIGURATION -->
<FaultConfiguration>
    <FaultSet>
        <Name>FC4</Name>
        <FaultMode>TRANSIENT</FaultMode>
        <Duration>4.0</Duration>
        <TriggerInstant>86.0</TriggerInstant>
        <Fault ref="_p5LWtRbKEeOw28kCcXd1kQ" />
    </FaultSet>
</FaultConfiguration>
<!-- FAULTS -->
<Faults>
    <Fault id="_p5LWtRbKEeOw28kCcXd1kQ">
        <Name>i4</Name>
        <Location ref="_LeWBYBbJEeOEgpzBm6gXmA" />
        <FaultEffect>Integer_Constant</FaultEffect>
        <ConstantValue>600</ConstantValue>
    </Fault>
</Faults>
<!-- FAULT LOCATIONS -->
<Locations>
    <Location id="_LeWBYBbJEeOEgpzBm6gXmA">
        <Component>system_railwayss.das_superv.das_odo.job_odo</Component>
        <PortType>Input</PortType>
        <Entity>DAS_ODO_ENCODER1</Entity>
    </Location>
</Locations>
```

location of the component that must simulate a faulty behavior. In the case of faults from the PIM library, which are related to signals, the name of the *entity* to be sabotaged and its *port type* (input or output) also need to be specified.

It is worth to note that the definition of a fault campaign is optional. In case no fault-campaign is specified at the beginning of the simulation, a fault-free simulation is driven by the ATE.

User Interface

The Fault Injection Unit provides the following instructions to enable the users to interact with the ATE via the Python shell:

- `void LoadFIDescriptionFile(string file)`: This command enables to specify which fault injection configuration is going to be loaded and executed by the TCI. Similarly to the TCMI, this command accepts both absolute and relative paths for the specification of the fault injection configuration file. This command might be called more than once, enabling the testing teams to run more than one fault injection campaign in a simulation. On the other hand, if the

`Confirm()` command is called without specifying any FI campaign file, the ATE will run a fault-free simulation.

- `void Confirm(void)`: This instruction is used to inform the FIU that the setup of the fault injection campaign is finished. When this command is sent to the FIU, the FIU notifies the ATE that it is ready to start. When the ATE receives the same notification from the other modules, the initialization phase finishes and the simulation begins. Therefore, it is mandatory to send this command in order to ensure the simulation will start.

Fault Configuration meta-model

Figure 5.4 shows the generic UML meta-model of the Fault Injection Configuration files.

Test Point Manager (TPM)

The Test Point Manager (TPM) is the module that enables to observe the evolution of the internal signals of the SUT as the time flows. The TPM is composed by the following three submodules (see figure 5.5):

- Test Point Parser
- Test Point Set
- Test Point Set Interpreter & Data Recorder

Similarly to the TCI and FIU, the configuration of the test points is read by the Test Point Parser and saved in the Test Point Set during the initialization phase. During simulation the TP Set Interpreter identifies each signal of the SUT and sends the data of the signals specified in the TP Set to the Data Recorder, which stores the values of each signal along with its time stamp. When the simulation finishes, the TPM creates a value-change-dump file (*.vcd) with all the data collected during simulation, in order to enable the test engineers to assess the emerging behavior of the system. The value-change-dump file format is an industrial standard format that captures information about the value changes of signals and waveforms. Several SW programs are capable of interpreting .vcd files, e.g., GTKWave [wwwk] (open source), EZWave (by Mentor Graphics) [wwwj], or SimVision (by Cadence) [wwwz].

Analogously to the FIU, the TPM reads the values of the internal signals of the SUT every time a job reads or writes a signal. Therefore, the specification of the test point locations, which is also defined in XML files, follows the next scheme: First, the file must specify the hierarchical location of the job that will read or write the signal. For each job, a set of entities (signal names) might be specified, distributed in groups

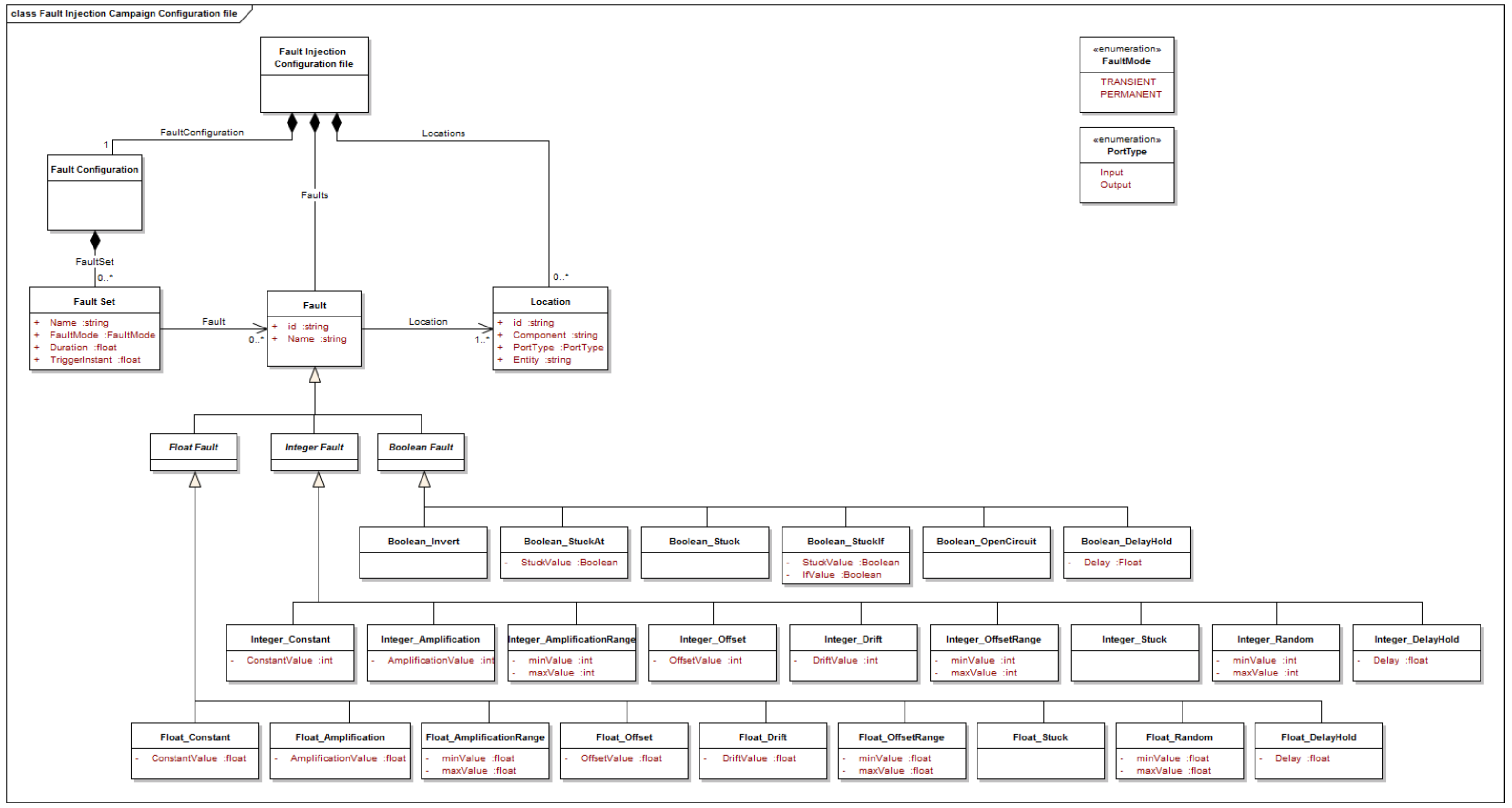


Figure 5.4: UML Meta-model of Fault Injection Configuration XML files

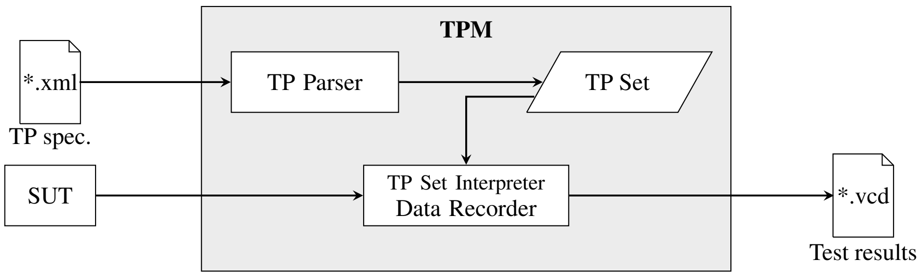


Figure 5.5: Test Point Manager Module

of inputs and outputs. The explicit specification of the job that will read or write the signal avoids ambiguities in the case in which a given name is repeated between ports of different jobs; moreover, specifying whether the signal is treated as an input or output by the job enables distinguishing between the reading and writing activities of a job, which might be useful in systems where a given job reads a variable at the beginning of its execution and re-writes the same variable at the end. Listing 5.3 shows an example of the syntax of these files.

User Interface

Users are allowed to interact with the Test Point Manager by the following commands:

- `void LoadTPDescriptionFile(string file)`: This command enables to establish the test-point specification file that will be loaded by the TPM, that is, enables specifying the signals that will be observed. Similarly to the other modules, the paths to the TP specification file can be specified as relative to the executable of the model or as absolute paths. This command might be called more than once; in that case, all signals specified in all the files would be recorded. On the contrary, if this command was not called before `Confirm()`, the ATE would run a simulation but no information about the internal signals would be stored.

- `void SetTraceFileName(string filename)`: This command enables specifying a name for the value-change-dump file provided by the TPM with the evolution of the values of the signals over the simulation. This command is mandatory unless no test-point specification file is stated for the simulation.

- `void Confirm(void)`: This instruction is used to inform the TPM that the setup of the test-point configuration is finished. When this command is sent to the TPM, it notifies the ATE that it is ready to start. When the ATE receives the same notification from the other two modules, the initialization phase finishes and the

Listing 5.3: Example of test points specification file (XML file)

```
<!-- TEST POINT LOCATIONS -->
<Locations>
    <!-- NODE_EVC_A -> PROC -> CORE1 -> JOB_ODOMETRY -->
    <Job hierarchy="systemTSS.clusterEVC.nodeEVCA.procA.core1.jobODO">
        <Inputs>
            <Entity>DAS_EVC_IN_ENC1</Entity>
            <Entity>DAS_EVC_IN_ENC2</Entity>
            <Entity>DAS_EVC_IN_ACCEL</Entity>
            <Entity>DAS_EVC_IN_BAL_NEWBAL</Entity>
            <Entity>DAS_EVC_IN_BAL_POS</Entity>
            <Entity>DAS_EVC_IN_BAL_INCL</Entity>
            <Entity>DAS_EVC_IN_BAL_NEXTPOS</Entity>
        </Inputs>
        <Outputs>
            <Entity>DAS_EVC_ST_S</Entity>
            <Entity>DAS_EVC_ST_V</Entity>
        </Outputs>
    </Job>
    <!-- NODE_VOT_A -> PROC -> CORE -> JOB_VOTER_A -->
    <Job hierarchy="systemTSS.clusterEVC.nodeVotA.procVotA.coreVotA.jobVotA">
        <Inputs>
            <Entity>CORE_VOTERA_IN_EMERG_A</Entity>
            <Entity>CORE_VOTERA_IN_EMERG_B</Entity>
            <Entity>CORE_VOTERA_IN_EMERG_C</Entity>
            <Entity>CORE_VOTERA_IN_SERV_A</Entity>
            <Entity>CORE_VOTERA_IN_SERV_B</Entity>
            <Entity>CORE_VOTERA_IN_SERV_C</Entity>
            <Entity>CORE_VOTERA_IN_WARN_A</Entity>
            <Entity>CORE_VOTERA_IN_WARN_B</Entity>
            <Entity>CORE_VOTERA_IN_WARN_C</Entity>
        </Inputs>
        <Outputs>
            <Entity>CORE_VOTERA_OUT_EMERG</Entity>
            <Entity>CORE_VOTERA_OUT_SERV</Entity>
            <Entity>CORE_VOTERA_OUT_WARN</Entity>
            <Entity>CORE_VOTERA_OUT_FAILURE</Entity>
            <Entity>CORE_VOTERA_OUT_SYSTEMFAILURE</Entity>
        </Outputs>
    </Job>
    <!-- NODE_VOT_B -> PROC -> CORE -> JOB_VOTER_B -->
    <Job hierarchy="systemTSS.clusterEVC.nodeVotB.procVotB.coreVotB.jobVotB">
        <Inputs>
            <Entity>CORE_VOTERB_IN_EMERG_A</Entity>
            <Entity>CORE_VOTERB_IN_EMERG_B</Entity>
            <Entity>CORE_VOTERB_IN_EMERG_C</Entity>
            <Entity>CORE_VOTERB_IN_SERV_A</Entity>
            <Entity>CORE_VOTERB_IN_SERV_B</Entity>
            <Entity>CORE_VOTERB_IN_SERV_C</Entity>
            <Entity>CORE_VOTERB_IN_WARN_A</Entity>
            <Entity>CORE_VOTERB_IN_WARN_B</Entity>
            <Entity>CORE_VOTERB_IN_WARN_C</Entity>
        </Inputs>
        <Outputs>
            <Entity>CORE_VOTERB_OUT_EMERG</Entity>
            <Entity>CORE_VOTERB_OUT_SERV</Entity>
            <Entity>CORE_VOTERB_OUT_WARN</Entity>
            <Entity>CORE_VOTERB_OUT_FAILURE</Entity>
            <Entity>CORE_VOTERB_OUT_SYSTEMFAILURE</Entity>
        </Outputs>
    </Job>
</Locations>
```

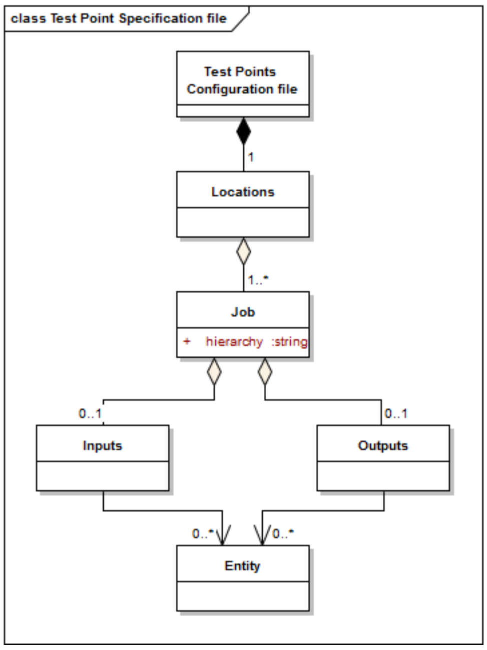


Figure 5.6: UML Meta-model of Test Point Configuration XML files

simulation begins. Therefore, it is mandatory to send this command in order to guarantee that the simulation will start.

Fault Configuration meta-model

Figure 5.6 shows the generic meta-model of the Test Point Configuration files in UML language.

5.2 Fault Injection Libraries

As mentioned in the previous section, the Fault Injection Unit of the ATE provides libraries of faults for both platform independent and platform specific models. This way, test engineers can invoke a fault by just specifying the name of the desired fault effect in the fault-configuration XML file, and they do not need to model the effect of the fault explicitly. This increases the usability of the PS-TTM ATE framework.

Fault Library for Platform Independent Models

Since platform independent models do not cover any aspect related to the target platform in PS-TTM, the library of faults focuses on faults at signal levels. This fault library draws on the failure mode functions (FMFs) defined in the MOGENTES project

[MOG09]. These failure modes represent the "effects of faults/errors that would lead to a failure in a system". The fault library for PIMs specifies faults for signals carrying boolean, integer and floating-point variables:

Fault Effects for Boolean Signals

- **Inversion**:
  - *Name of the effect*: `Boolean_Invert`
  - *Configuration parameters*: None
  - *Description*: The boolean value of the signal is inverted (i.e., it is changed from 'false' to 'true' or vice-versa).
  - *Usage*: This fault effect can be used to simulate bit flips in memory cells, such as Single Event Upsets (SEUs) induced by energetic particles.
- **Stuck At**:
  - *Name of the effect*: `Boolean_StuckAt`
  - *Configuration parameters*:
    - `StuckValue`: *Boolean*
  - *Description*: The signal gets stuck at the value given by the `StuckValue` parameter, which must be set to either 'false' or 'true'.
  - *Usage*: The 'Stuck At' fault effect is used to reproduce a failure in a memory cell or connector that provokes a given data to stuck at a constant value.
- **Stuck**:
  - *Name of the effect*: `Boolean_Stuck`
  - *Configuration parameters*: None
  - *Description*: The signal gets stuck at the value the variable had at the time step in which the fault was injected.
  - *Usage*: 'Stuck' can be used to reproduce the effect of a communication loss between two components, what would result in keeping the value of the variable unaltered.
- **Stuck If**:
  - *Name of the effect*: `Boolean_StuckIf`
  - *Configuration parameters*:
    - `StuckValue`: *Boolean*

    * `IfValue`: *Boolean*

  - *Description*: In case the signal has the value defined by the `IfValue` parameter at the instant at which the fault is injected, it gets stuck at the value defined by `StuckValue`. Else, the signal does not get stuck and thus no fault is injected.
  - *Usage*: This fault effect can be used to simulate different phenomena also reproduced by 'Stuck' and 'Stuck At' effects. In addition, 'Stuck If' gives a bigger freedom to the test developers to control the activation of the fault by choosing a required pre-condition for it.

- **Open Circuit**:
  - *Name of the effect*: `Boolean_OpenCircuit`
  - *Configuration parameters*: None
  - *Description*: The signal behaves as a non-connected wire, i.e., it might take any value arbitrarily.
  - *Usage*: This fault effect is used to simulate different situations in which a signal can get a random value, such as a broken wire, a defective connection or a communication through a very noisy environment.

- **Delay**:
  - *Name of the effect*: `Boolean_DelayHold`
  - *Configuration parameters*:
    * `Delay`: *Float*
  - *Description*: The sequence of values of the signal is delayed by the amount of time defined by the `Delay` parameter. When the fault is injected, the value of the signal gets stuck until the `Delay` time has passed, and the fault injector starts storing the sequence of values of the signal in an internal FIFO buffer. Once the `Delay` time finishes, the fault injector begins to forward values of its FIFO buffer to the signal at each time step. The fault injector remains buffering the sequence of values received from the signal and returning the old values to the signal until the fault injection activity finishes, i.e., until the simulation finishes in the case of permanent faults, or until the `Duration` specified in the fault configuration is completed.
  - *Usage*: This fault effect can be used to simulate delays introduced by a data-traffic overhead in the communication system or by the excessive length of wires.

Fault Effects for Integer & Floating Point Signals

- **Constant**:
  - *Name of the effect*: `Integer_Constant` / `Float_Constant`
  - *Configuration parameters*:
    - `ConstantValue`: *Integer / Float*
  - *Description*: The value of the signal gets stuck at the value given by the `ConstantValue` parameter.
  - *Usage*: This fault effect can be used to simulate several different situations, such as a broken encoder that keeps providing a constant value to the system, a faulty sensor measuring a constant value, or a rotational component that is blocked by an external object.
- **Amplification**:
  - *Name of the effect*: `Integer_Amplification` / `Float_Amplification`
  - *Configuration parameters*:
    - `AmplificationValue`: *Integer / Float*
  - *Description*: The value of the signal gets multiplied by a the value provided by the `AmplificationValue` parameter.
  - *Usage*: The amplification fault effect can simulate a number of faults, e.g. erroneous sensor positioning and/or orientation, wrong parametrization of components, bugs due to misunderstanding of paramenter units, etc.
- **Amplification Range**:
  - *Name of the effect*: `Integer_AmplificationRange` / `Float_AmplificationRange`
  - *Configuration parameters*:
    - `minValue`: *Integer / Float*
    - `maxValue`: *Integer / Float*
  - *Description*: The value of the signal is amplified by a value randomly selected from the interval given by the user, i.e., [`minValue`, `maxValue`].
  - *Usage*: This fault effect is mainly used to introduce noise in signals.
- **Drift**:
  - *Name of the effect*: `Integer_Drift` / `Float_Drift`

  - *Configuration parameters*:
    * `DriftValue`: *Integer / Float*
  - *Description*: The value of the signal drifts away from its real value, by incrementing the previous value of the signal with the value specified by the `DriftValue` parameter at each time step.
  - *Usage*: This effect is used to simulate incremental faults in components, such as a failure in a counter that causes it to count more events than it should at each iteration.

- **Offset**:
  - *Name of the effect*: `Integer_Offset` / `Float_Offset`
  - *Configuration parameters*:
    * `OffsetValue`: *Integer / Float*
  - *Description*: The fixed value provided as the `OffsetValue` is added (or substracted if negative) to the actual value of the signal, thus introducing an offset to the actual value of the signal.
  - *Usage*: This fault effect can be used to reproduce situations in which a sensor has not been correctly calibrated.

- **Offset Range**:
  - *Name of the effect*: `Integer_OffsetRange` / `Float_OffsetRange`
  - *Configuration parameters*:
    * `minValue`: *Integer / Float*
    * `maxValue`: *Integer / Float*
  - *Description*: A value randomly selected between the [`minValue`, `maxValue`] set is added to the actual value of the signal.
  - *Usage*: This fault effect is mainly used to introduce noise in signals.

- **Stuck**:
  - *Name of the effect*: `Integer_Stuck` / `Float_Stuck`
  - *Configuration parameters*: None
  - *Description*: The signal gets stuck at the value the variable had at the time step in which the fault was injected.

  - *Usage*: This fault effect can be used to simulate several different situations, such as a broken encoder that keeps providing a constant value to the system, a faulty sensor measuring a constant value, or a rotational component that is blocked by an external object.

- **Random**:
  - *Name of the effect*: `Integer_Random` / `Float_Random`
  - *Configuration parameters*:
    - `minValue`: *Integer / Float*
    - `maxValue`: *Integer / Float*
  - *Description*: The signal takes an arbitrary value from the set specified by [`minValue`, `maxValue`].
  - *Usage*: This fault effect is used to reproduce situations in which a signal can get a random value, such as a broken wire or a defective electrical connection.

- **Delay**:
  - *Name of the effect*: `Integer_DelayHold` / `Float_DelayHold`
  - *Configuration parameters*:
    - `Delay`: *Float*
  - *Description*: The sequence of values of the signal is delayed by the amount of time defined by the `Delay` parameter. When the fault is injected, the value of the signal gets stuck until the `Delay` time takes over, and the fault injector starts storing the sequence of values of the signal in an internal FIFO buffer. Once the `Delay` time finishes, the fault injector begins to forward values of its FIFO buffer to the signal at each time step. The fault injector remains buffering the sequence of values received from the signal and returning the old values to the signal until the fault injection activity finishes, i.e., until the simulation finishes in the case of permanent faults, or until the `Duration` specified in the fault configuration is completed.
  - *Usage*: This fault effect can be used to simulate delays introduced by a data-traffic overhead in the communication system or by the excessive length of wires.

The fault library natively provided by the PS-TTM ATE for platform independent models covers a wide range of failures that are likely to occur in systems, and provides a straightforward and easy way to simulate them at a platform independent level. However, in case it is needed, this library might be extended to new fault models.

Table 5.1 summarizes the fault library for platform independent models.

Table 5.1: Fault library for platform independent models

	Fault Effect	Config. parameters	Description
Boolean	Invert	-	Boolean value is inverted
	Stuck At	stuck_value	Signal gets stuck at a given value
	Stuck	-	Signal gets stuck at the actual value
	Stuck If	stuck_value, condition	Signal gets stuck if a given condition holds
	Open Circuit	-	Wire is disconnected, signal takes an arbitrary value (noise)
	Delay	delay	Signal is delayed by an amount of time
Integer / Float	Constant	constant_value	Signal gets stuck at a given constant value
	Amplification	ampl_value	Signal is amplified by fixed value
	Amplification Range	min_amp_value, max_ampl_value	Signal is amplified by a randomly selected value (between given max. and min. values)
	Drift	drift_value	At each time stepc, the signal drifts away from its nominal value by a given value
	Offset	offset_value	A given fixed offset is added to the signal
	Offset Range	min_offset_value, max_offset_value	A randomly selected offset value is added to the signal (between given max. and min. values)
	Stuck	-	Signal gets stuck at the actual value
	Random	min_value, max_value	Signal takes an arbitrary value (between given max. and min. values)
	Delay	delay	Signal is delayed by an amount of time

Fault Library for Platform Specific Models

As explained in section 4.5, the platform specific models of the PS-TTM rely on HW components specified at a high abstraction level. Hence, the fault library for PSMs is composed by the high-level effects to which HW-related faults have been typically reduced in the literature. This leads to a fault library composed by four main fault effects. However, since it is possible to extend the platform specific component library with more detailed models of HW components, the fault library may also be extended in order to perform fault injection at a lower level of abstraction, which would give a bigger level of control to the test engineers for the injection of faults at HW components.

Even though the faults provided by the PSM fault library refer to faults in HW components, as mentioned before, the PS-TTM ATE performs fault injection during the communication phases of the simulation. Therefore, the ATE emulates faulty HW components by sabotaging all their outgoing signals according to the fault effect specified in the fault configuration. In other words, from the perspective of the PS-TTM ATE, a faulty HW component is a black box whose push activities are all sabotaged, thus giving the illusion of being a faulty component. This approach provides an equivalent effect to modifying the HW component models in order to simulate an erroneous behavior. Nevertheless, it has the advantage that it is a non-intrusive approach, i.e., it does not require to perform any modification to the model, such as substituting the correct component by an erroneous one (mutant based fault injection).

The fault library for PSMs is composed by the following effects:

- **Corruption**:
    - *Name of the effect*: `HW_Corruption`
    - *Configuration parameters*: None
    - *Description*: The data provided by the interfaces of the HW component is corrupted.
    - *Usage*: This fault effect enables to reproduce a situation in which the functionality of the hardware component performs incorrectly. Thus, it can be used to simulate different situations that can provoke a faulty behavior in the value domain, such as noisy environments or defective electrical connections.
- **No execution**:
    - *Name of the effect*: `HW_NoExecution`
    - *Configuration parameters*: None
    - *Description*: The functionality of the HW component is not executed. No data is provided in the output interfaces.
    - *Usage*: This fault effect can be used to simulate errors caused by faults in power supplies, cuts in wires, or misbehaviors of HW components due to corrupted or incorrect data.
- **Out of time**:
    - *Name of the effect*: `HW_OutOfTime`
    - *Configuration parameters*:
        - `Delay`: *Float*
    - *Description*: The time bounds of the functionality are not respected, i.e., data is provided later than expected. The `Delay` parameter establishes the width of the time-lapse. The fault is injected as follows: when the fault is triggered, the value of the signals provided by the HW component get stuck until the `Delay` time takes over, and the fault injector begins to store the sequence of values of each of the signals in a dedicated FIFO buffer. Once the `Delay` time finishes, the fault injector begins to forward values of each buffer to its corresponding signal at each time step.
    - *Usage*: The ‘Out of Time’ fault effect can be used to introduce a delay in the response of a HW component, which reproduces the effect of a number of anomalies in the system, such as an overload of the CPU, an excess of traffic in the communication system, or any other timing-related error caused by a misbehavior of a component.

Table 5.2: Fault library for PSM models

Fault Effect	Conf. parameters	Description
Corruption	-	The functionality is performed incorrectly. The data provided by the output interface is corrupted.
No execution	-	The functionality is not executed. No information is provided as a result.
Out of time	Delay	Time bounds of the functionality are not respected. Data is provided later than expected.
Babbling	Delay	Data in the interface is erroneous both in terms of content and time.

- **Babbling**:
  - *Name of the effect*: `HW_Babbling`
  - *Configuration parameters*:
    - `Delay`: *Float*
  - *Description*: The data provided by the output interfaces of the HW component is incorrect both in terms of content and time. This fault effect can be understood as a combination of the 'Corruption' and 'Out of Time' faults, where the `Delay` parameter again establishes the width of the time-lapse.
  - *Usage*: This fault effect can be used to simulate environments with very unfavorable conditions, such as those including big levels of noise (which would lead to corruption of data) and overhead of data-traffic in communications (which would cause the system to miss required deadlines).

As mentioned previously, the library of faults for platform specific models can be extended with more detailed models in case the designers specified lower level HW components in their designs. In addition to that, platform specific models can make use of the library of faults for the platform independent models, which enables to validate the PSMs against lower-level faults. As a consequence of this compatibility between PSMs and the PIM-related fault library, the fault campaigns used during the verification of a system at a platform independent level can be re-used for the verification of the PSM of the same system, in order to guarantee that the functional properties of the system were not compromised when transforming the PIM into the PSM.

Since the fault effects defined in the library of faults for platform specific models refer to faults in HW components, the specification of a *PortType* and an *Entity* for this type of faults gets meaningless. Therefore, the FIU ignores the two mentioned fields during the construction of the fault injection set. In fact, if a fault location is only referenced by PSM related fault effects in a fault campaign, these parameters can be left undefined in the fault configuration file, as Listing 5.4 shows.

Table 5.2 provides a brief summary of the fault library for platform specific models.

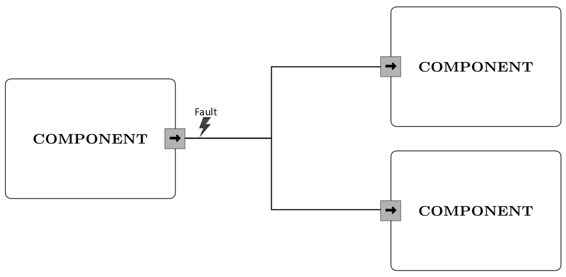
Listing 5.4: Example of fault configuration file with HW related fault

```
    <!-- FAULT CONFIGURATION -->
    <FaultConfiguration>
    <FaultSet>
        <Name>FC37</Name>
        <FaultMode>PERMANENT</FaultMode>
        <Duration></Duration>
        <TriggerInstant>120.0</TriggerInstant>
        <Fault ref="hV4tRVQKifNGh5L1vztK"/>
    </FaultSet>
</FaultConfiguration>
<!-- FAULTS -->
<Faults>
    <Fault id="hV4tRVQKifNGh5L1vztK">
        <Name>outoftime6</Name>
        <Location ref="ZjYD4BVP03DD4Tbw0Uon"/>
        <FaultEffect>HW_OutOfTime</FaultEffect>
        <Delay>0.50</Delay>
    </Fault>
</Faults>
<!-- FAULT LOCATIONS -->
<Locations>
    <Location id="ZjYD4BVP03DD4Tbw0Uon">
        <Component>system_railwayss.clrail.nodeevcC.procevc.coreA</Component>
        <PortType></PortType>
        <Entity></Entity>
    </Location>
</Locations>
```

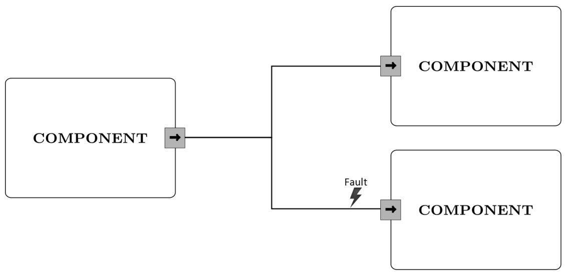
5.3 Symmetric and Asymmetric Fault Injection

When a signal provided by a job is forwarded to more than one component, it has to be replicated in different channels. In case of a fault in such a signal, the consistency between the values read by the receiving components might be compromised. When the incorrect service is equally perceived by all the consumers, the failure is considered consistent; on the contrary, if some of the receivers perceive differently incorrect service, the failure is called inconsistent failure or more frequently 'byzantine failure' [LSP82]. Byzantine failures are the most difficult ones to handle, since they have the potential to confuse the correct components. In fact, in extreme cases, a receiver might classify the failing component as erroneous whereas another receiver might identify it as correct, thus leading to an inconsistent view of the failed component among the correct components.

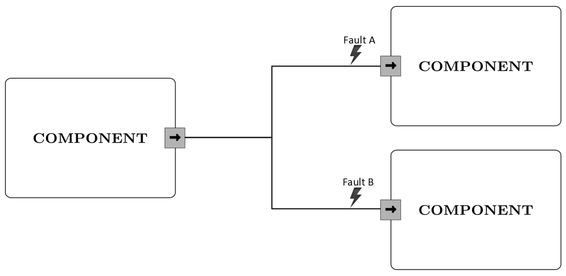
[Kop11] provides a good example of the danger of byzantine failures. Assume a system in which a sender outputs a voltage value to several receivers through a bus. In case the voltage of the high-level output of a sender is slightly below the level specified for the high-level state, then some receivers might still accept the signal, assuming the value of the signal is high, while others might not accept the signal, assuming the value



(a) Symmetric Fault Injection



(b) Asymmetric Fault Injection (Inconsistent classification of erroneous component)



(c) Asymmetric Fault Injection (Inconsistent classification of failure)

Figure 5.7: Symmetric and Asymmetric Fault Injection

is not high. This leads to an inconsistent view of the system among the receivers, which are correct.

As explained in the previous sections, the FIU of the PS-TTM ATE enables the testing teams to inject faults in any signal of the system under test. This fact opens the possibility to perform both symmetric and asymmetric fault injection in the models, depending on if the fault is injected at the instant in which a signal is being sent or when it is being received, as figure 5.7 ilustrates:

The ability to perform asymmetric fault injection opens the possibility to reproduce byzantine failures in the system under test, and therefore straightforwardly assess the fault-tolerance provided by the system against byzantine faults.

CHAPTER 6

Tooling

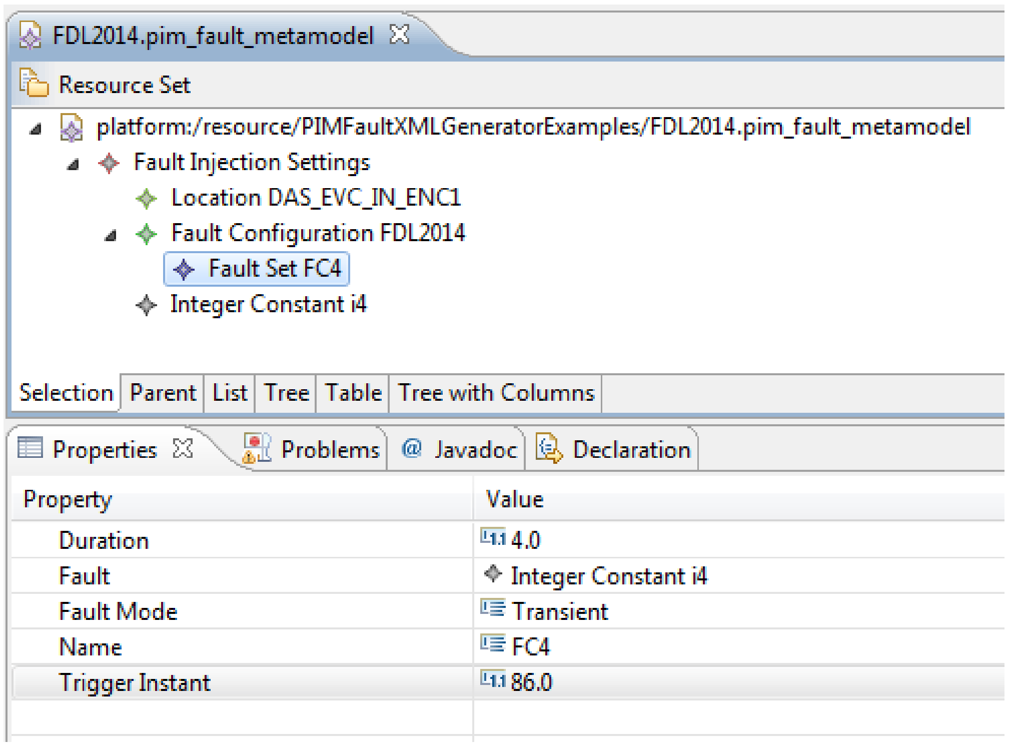
The PS-TTM simulation engine has been built as a library that extends SystemC with a set of macros and mechanisms that helps the designers in the definition of the models. Besides this, a set of tools has been developed in order to assist the designers during the development and testing processes of the systems. This chapter introduces the tools developed during this research work.

6.1 Graphical SFI Campaign Designer Tool

Manual generation of complete Simulated Fault Injection campaigns in textual XML specifications might be a cumbersome and error-prone task for users of a new testing system. Therefore, in order to ease the generation of SFI campaigns, a graphical SFI campaign-specification tool has been developed. Figure 6.1 shows a screenshot of the tool.

The tool, which has been built as a plugin for eclipse has been developed using EMF/Ecore, following the meta-model of the SFI campaigns shown in Figure 5.4. Thus, the tool enables to define a set of locations and faults, and then build the desired SFI configuration by defining a number of fault sets to be applied to the simulation.

The tool includes a code generator for the SFI campaigns, which automatically generates an XML-file corresponding to the SFI designed by the test developers. This way, hand coding of XML files is avoided and the execution of the campaigns designed with the tool can be performed in a transparent way for the user, avoiding errors and easing the definition of such campaigns.


FDL2014.pim_fault_metamodel

Resource Set

platform:/resource/PIMFaultXMLGeneratorExamples/FDL2014.pim_fault_metamodel
- Fault Injection Settings
  - Location DAS_EVC_IN_ENC1
  - Fault Configuration FDL2014
    - Fault Set FC4
  - Integer Constant i4

Property	Value
Duration	4.0
Fault	Integer Constant i4
Fault Mode	Transient
Name	FC4
Trigger Instant	86.0

Figure 6.1: Simulated Fault Injection Configuration Tool

6.2 Test Case Generator Script

Manual generation of test cases in a self-defined language might be an unmanageable task, especially if the system under test is not restricted to digital signals. For example, if a model under test requires a value of temperature as an input, manually specifying the evolution of the temperature over time at each time stamp would be infeasible. Thus, it would be reasonable to generate a model of a heater/cooler and simulate it in order to get a progressive evolution of the temperature over time.

Since the PS-TTM runs on SystemC and it is fully compatible with it, it offers the possibility to model the environment running in continuous time directly in SystemC-AMS, and simulate the model under test and the plant simultaneously. However, experience shows that this might increase the simulation time dramatically, particularly if the environmental model is not very simple.

Hence, it is a common practice in industry to design test campaigns by the implementation of test harnesses in dedicated tools that rely on the continuous MoC, such as MATLAB/Simulink [wwwx], Spice [Nag75] or Dymola [wwwh], and run the environmental model until it leads to the desired state.

Thus, in order to enable this practice for simulations with the PS-TTM-ATE, a test case generator script has been developed within the scope of this work. The test-generator tool, which has been developed as a script in Python, basically takes the output results recorded by a MATLAB/Simulink model simulation, and generates a PS-TTM compliant XML file with the corresponding set of inputs. This way, the automatically generated XML file can be directly read by the Test Case Interpreter of the PS-TTM ATE, which will run the desired test case against the PS-TTM model.

6.3 Test Result Interpretation Scripts

In order to assess the behavior of the model under test, the Test Point Manager of the PS-TTM ATE provides the results of the simulations as a value-change-dump file (*.vcd). The value-change-dump file format is an industrial standard format that captures information about the value changes of signals and waveforms. The '.vcd' format can be interpreted by a number of SW programs, such as GTKWave [wwwk] (open source), EZWave (by Mentor Graphics) [wwwj], or SimVision (by Cadence) [wwwz].

However, in spite of being a standard, it is not very widely accepted by the SW community. In fact, proprietary tools with capabilities to run '.vcd' files such as EZWave (by Mentor Graphics) [wwwj], or SimVision (by Cadence) [wwwz], give preference to their own proprietary formats, and offer limited functionality to '.vcd' files.

Besides, the most extended open source tool for vcd files, GTKWave [wwwk], provides rather limited graph visualization options. The visualization options are specially poor for non-digital signals.

Therefore, in order to extend the visualization capabilities and options of the simulation results of the PS-TTM models, a vcd-to-csv file translator has been created during this work. Comma-Separated-Values files (*.csv), are files that store tabular data (numbers and text) in plain-text form, and are more common than vcd files. CSV files can be read by most of the spreadsheet application SW on the market, such as Microsoft Office Excel [wwwp] and LibreOffice Calc [wwwn], or even by numerical computing SW, such as MATLAB [wwwo]. These tools provide a huge number of diagram visualization options and data transformation facilities. Thus, the vcd-to-csv translator greatly improves the usability of the simulation results provided by the PS-TTM ATE.

6.4 Graphical Modeling Tool (alpha version)

Typing a whole time-triggered system design in plain text might become a tedious work, particularly if the system is complex and has many interconnections. Thus, being able to build the structure of the system by means of a graphical modeling tool would relax the complexity of the design for the developers.

With this goal in mind, the PS-TTM will contain a graphical front end. The graphical user interface, which is currently under development, includes graphical modeling capabilities for both PIM and PSM models. PIM models are built hierarchically, by composition of Systems, DASes and jobs, which are depicted as blocks.

The generation of the PSM model is guided by the graphical tool in several manners. First the user must define the target platform. Then, the GUI recalculates the temporal properties of jobs, according to their previous specification in the PIM, and enables the user to deploy them into the platform components. The temporal properties of jobs might be modified by users if necessary.

Obviously, the graphical modeling tool includes automatic code generation capabilities for both PIM and PSM models, in order to perform the required simulations and fault injection experiments for their verification.

6.5 Integration of tools in the overall work-flow

Figure 6.2 shows how the previously described tools are integrated into the work-flow specified by the PS-TTM approach.

As the figure shows, the design of the system should start with the specification of the functional and non-functional requirements with a dedicated requirement management tool. Once the requirements have been identified, the testing teams can use the graphical SFI campaign designer tool to define their SFI campaigns, and generate the necessary environmental models with Simulink. When these tasks are finished, the SFI XML code generator and test case generator script will automatically generate the corresponding XML files for the evaluation of the system.

Simultaneously, the system designers can start the design of the PIM with the graphical modeling tool, and generate the textual PIM with the automatic code generator. The PS-TTM ATE will take the SFI campaign, test cases and test-point configuration file (developed manually), and perform the specified simulations by means of the PI-TTM execution engine.

The results of the simulation can then be immediately translated into CSV files by means of the vcd-to-csv translator. The test designers might then use any csv-compliant software to analyze and validate the results, and suggest any modification to the system in case the system does not fulfill any of the requirements.

If the PIM model is considered correct, it must be deployed into the platform model by means of the graphical tool, and automatically generate the textual version of the PSM. The PS-TTM ATE can then be used to execute the model against the test campaigns defined from the functional and non-functional requirements, and the test developers can again use the vcd-to-csv translator in order to validate the results with their favorite csv-compliant software.

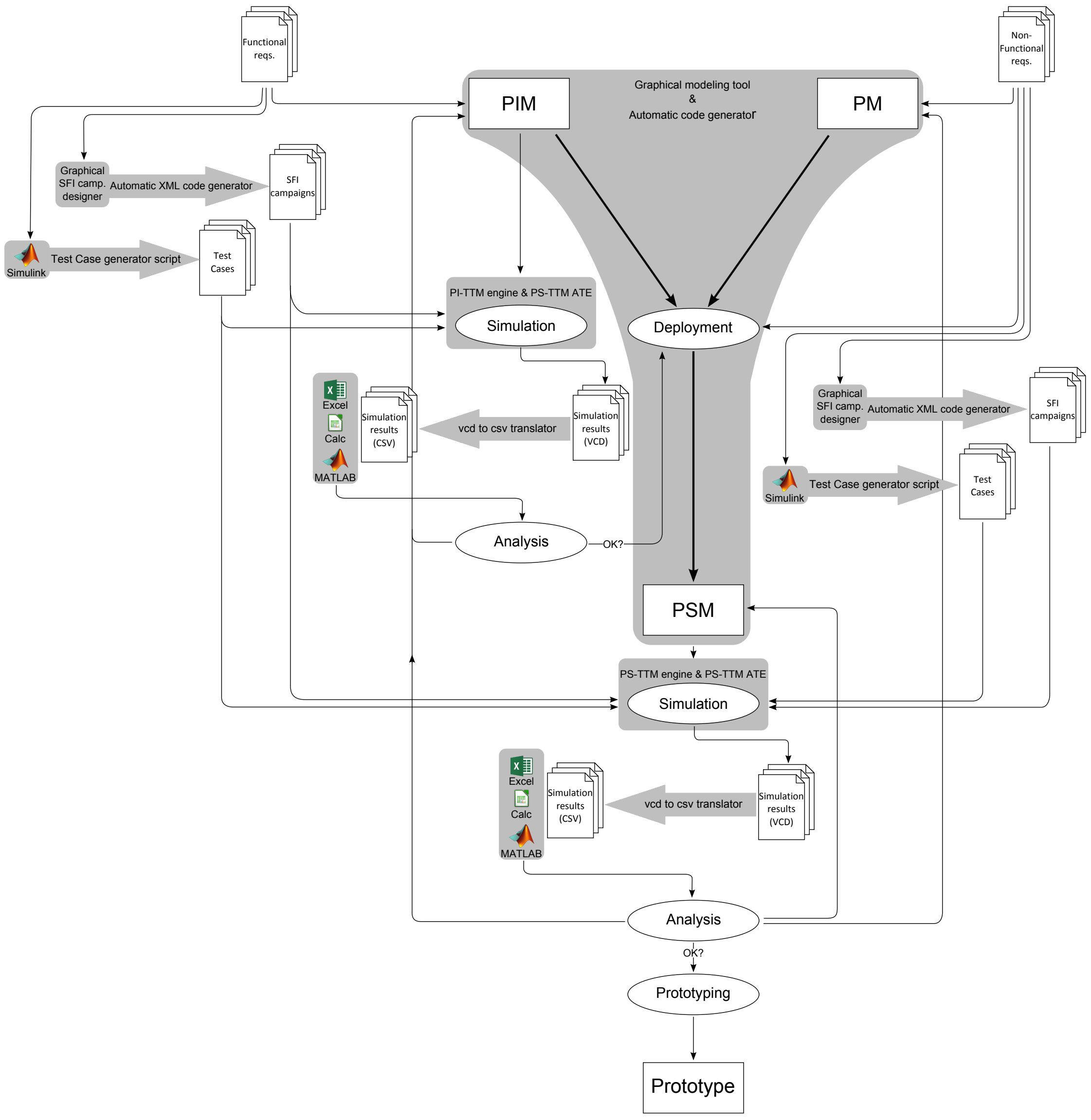


Figure 6.2: Integration of tools in the PS-TTM work-flow

CHAPTER 7

Case Study

This chapter addresses the evaluation of the proposed PS-TTM modeling framework and the PS-TTM Automatic Test Executor described in chapters 4 and 5 respectively, by means of a case study consisting on a safety-critical railway signaling system [Per11], and describes and analyzes the results obtained in the simulations.

7.1 European Train Control System (ETCS)

This section describes the European Train Control System (ETCS), which has been used as a case study for the assessment of the proposed approaches.

The ETCS constitutes the on-board unit of the European Railway Traffic Management System (ERTMS), a European Union backed initiative for the definition of a unique train signaling standard throughout Europe [WGR09]. The ETCS, prevents over-speeding in high-speed trains by supervising the traveled distance and speed and activating an emergency brake when the train exceeds the authorized values. The safety requirements for the ETCS state that it shall be designed as a safety-critical embedded system for safety integrity level 4 (SIL-4). In case a massive system failure occurs, the system should reach the safe state in which the emergency brakes are applied and the train is stopped. Thus, the system is classified as a fail-safe system.

Architecture of the ETCS

As Figure 7.1 illustrates, the ETCS is composed by several subsystems connected to the central safety processing unit called European Vital Computer (EVC):

- European Vital Computer (EVC): is the locomotive central safety processing unit that communicates with all subsystems and executes all safety functions associ-

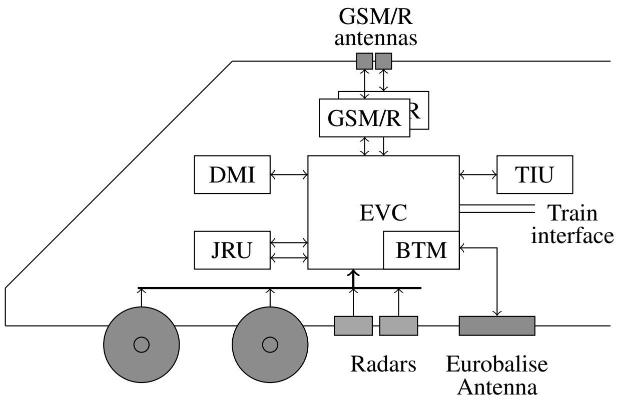


Figure 7.1: ETCS on-board reference architecture

ated to the traveling speed and distance supervision. The EVC executes the safety kernel and includes the odometry subsystem, which estimates the distance traveled by the train and its speed based on a set of diverse sensors.

- Driver Machine Interface (DMI): interface for the driver of the train, which is periodically updated with state parameters such as traveling speed and position of the train, and transmits sporadic event information (e.g., button pressed).

- Juridical Recorder Unit (JRU): subsystem responsible of recording all relevant external events (e.g., new eurobalise message) and internal events (e.g., activate emergency brake).

- Balise Transmission Module (BTM): this unit receives and interprets all the information provided by the eurobalises as the train passes them, and transmits it to the EVC. The Loop Transmission Module (LTM) provides analogous functionality with the data received from Euroloops.

- Global System for Mobile Communications - Railway (GSM/R): is an interface for the management of bidirectional information exchange between the remote control centers and the train.

- Train Interface Unit (TIU): reads / writes a set of input / output digital values, such as the emergency brake digital output.

- Odometry Sensors: is a group of sensors consisting on encoders, Doppler radars and longitudinal accelerometers that provide a set of measurements for angular speed and acceleration, and send the measurement data to the EVC.

Operating levels

The ETCS is specified to operate on three infrastructure deployment levels:

- ETCS Level 1: Eurobalises and optionally Euroloops are used by the infrastructure to communicate with the train via BTM and LTM, providing absolute position and track condition information. The Eurobalises are standardized components that are placed in specific locations of the railway. Every time the train passes an eurobalise, the balise transmits a telegram with its data. The BTM captures the telegram and transmits it to the EVC. The Euroloops work in a similar way to the Eurobalises, but they are actually built as a cable that is installed along the railway. This way, they extend the contact range of Eurobalises by providing a semi-continuous signal transmission along up to 800 meters. Level 1 ETCS systems do not include GSM/R communication.

- ETCS Level 2 and 3, extend previous level 1 by a GSM/R communication system that provides all track and operation related data. Eurobalises are used for odometric purposes, they provide absolute position information.

Functionality of the EVC

The functionality of the simplified version of the EVC used in this case study can be summarized in 4 main tasks: estimation of the position and speed, control of the operational mode, control of the emergency brake, and control of the service brake and warnings.

- Speed and position estimation: This task is performed by the odometry subsystem. In order to perform the estimation, the odometry subsystem gets the measurements provided by the angular speed encoders located in the wheels of the train and a longitudinal accelerometer placed in the chassis. Besides, in case a BTM detects a balise in the railway, the odometry subsystem reads its information and overwrites its estimated position with the one provided by the balise. According to the current standard, the maximum error made in the estimation of the position should not exceed five percent of the traveled distance plus 5 meters, at a maximum speed of $500km/h$. Equation 7.1 shows the accuracy required by the standard, where $s_m(t)$ represents the position estimated by the odometry system at instant t, and $s(t)$ represents the actual position of the train at instant t:

$$\forall t, |s_m(t) - s(t)| \leq 5m + (5/100) \cdot s(t) \tag{7.1}$$

- Operational mode control: This task must activate the *Standby* or *Supervision* operational modes depending on the command sent by the driver via the DMI.

The active mode is sent both to the Emergency and Service brake control units. The *Standby* mode is supposed to be active when the train is stopped, since it activates the emergency brake and deactivates the warnings and service brakes. In *Supervision* mode, the EVC supervises the current speed and position of the train and activates the warning and brakes when the maximum permitted speed values are exceeded.

- Emergency brake control: The emergency brake control unit implements the safety-critical (SIL-4) functionality of the system. To do so, it receives the information about the position and speed estimated by the odometry system, the *Standby* and *Supervision* activation signals from the mode control unit, and the *reset* command from the DMI.

  In case the operational mode control unit sets the system to *Standby* mode, this task activates the emergency brake.

  On the contrary, if the system is set to *Supervision* mode, the emergency brake control compares the estimated distance and speed to a pre-defined braking-curve that sets a maximum speed for each point in the track. If the estimated speed is higher than the maximum authorized speed, the emergency brake is activated and blocked. In order to release the emergency brake, the train must be stopped and and the driver must send a *reset* command through the DMI.

- Service brake control: The service brake control unit implements the non-safety-critical functionality of the EVC. Its functionality is similar to the emergency brake control, but it manages a light alarm, which warns the driver of an over-speed of the train, and the service brake, which lowers the speed of the train in case the over-speed remains.

  If the system is in *Standby* mode, the service brake and the warning are deactivated.

  However, in *Supervision* mode, the warning signal and service brake are activated when the speed of the train reaches the warning activation speed and the service brake activation speed respectively. The maximum speeds are pre-defined in two braking-curves that define maximum speeds for warning and service brake activation at each point in the track. Both the warning and the service brake are deactivated when the speed of the train falls below the warning activation speed.

7.2 Modeling the system

This section describes the process followed for the modeling of a simplified version of the ETCS by means of the PS-TTM modeling approach presented in this work. For the

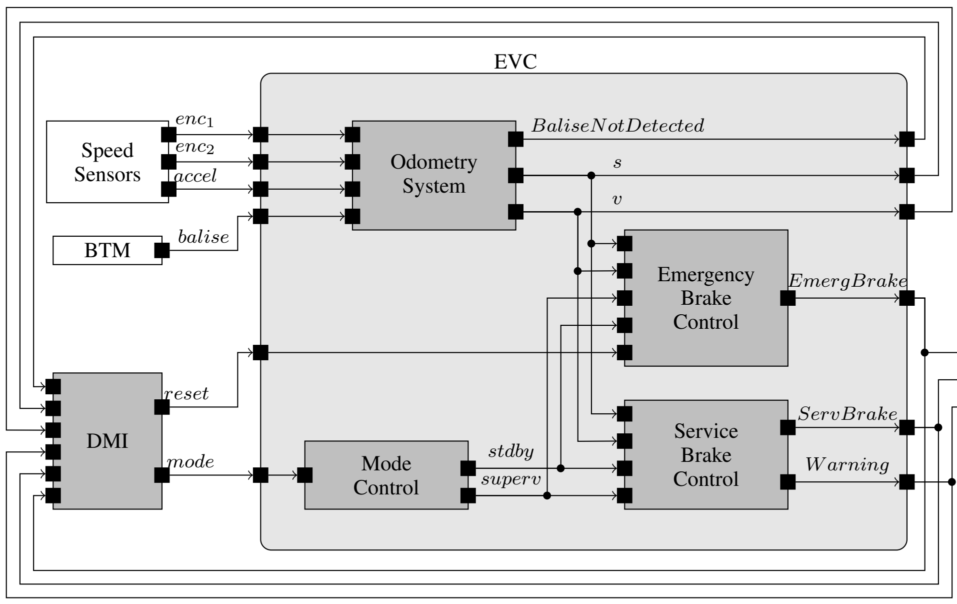


Figure 7.2: Case Study: Functionality of the PIM

sake of simplicity, we decide to omit the GSM/R and JRU subsystems from the design of the case-study. The case-study focuses on the design of the EVC and the DMI, which are treated as the system under test. Thus, we consider the TIU, odometry sensors and BTM are the environment of the system.

Platform Independent Model

As explained in chapter 4, the work-flow proposed by the PS-TTM commits the engineers to start the development of their intended system by first designing its PIM. The PIM should specify the functional behavior of the system.

Figure 7.2 shows a graphical overview of the functional interconnections between the tasks.

As described in the specification of the ETCS, the functionality of this system under test can be described as a set of five tasks (4 for the supervision and 1 for the DMI).

- Supervision tasks:
  - Odometry task: The odometry subsystem reads the data provided by the sensors and the balise transmission module. With that information, it provides an estimation of the position and speed of the train to the emergency brake and service brake controls. In addition, it sends a warning to the DMI in case an expected balise is not detected in the railway.

  - Mode control: Receives the operational mode selection command from the driver interface, and selects the operational mode of the train, which is sent to the emergency brake and service brake controls.
  - Emergency brake control: Decides whether the emergency brake has to be activated, depending on the operational mode control, position and speed of the train and the reset command received from the DMI.
  - Service brake control: Activates or deactivates the service brake and overspeed warning depending on the mode control in which the train is operating and its position and speed.

- DMI task: Enables the driver to activate the *reset* signal and choose between the two operational modes. The reset command is directly sent to the Emergency brake control task, whereas the selection of the mode is provided to the operational mode control. In addition, it displays information for the driver, including the position and speed of the train, state of emergency brake, service brake, and overspeed warning.

Thus, following the directives of the PS-TTM, we describe the functionality of the system under test in a hierarchical model. In this case, and in agreement with the spatial distribution of each of the tasks described above, we design the PIM of our system in PS-TTM as a model consisting on two main Distributed Application Subsystems (DASes), the *DMI* DAS and the *SupervisionSubsystem* DAS, as Figure 7.3 illustrates. The *SupervisionSubsystem* DAS is composed by three sub-DASes that contain four jobs, one for each of the supervision tasks described previously. The different interfaces and i-components that are automatically included by the PS-TTM modeling framework have been omitted from the figure for the sake of clarity. The components shaded in gray are automatically given by the PS-TTM library.

As stated previously, the system activates an emergency brake when the values estimated by the odometry system exceed the authorized limits. Therefore, the odometry algorithm must provide accurate and reliable measurements, which must not exceed the maximum error tolerated by the standards even in the case of faults in the system (Equation 7.1). Thus, the algorithm is usually based on a fault-tolerant sensor-fusion approach. In this case, we design the algorithm following one of the approaches described by Malvezzi et al. in [MAR10]. The algorithm estimates the speed of the train and the traveled distance with the information provided by an accelerometer that measures the acceleration of the train and two encoders that measure the speed of a different wheel each, as figure 7.2 showed.

Listing 7.1 shows the way in which the PIM represented by Figure 7.3 is defined in the PS-TTM. This listing shows the general structure of the PIM of our case study in PS-TTM. The fact that jobs are treated as atomic components by the PS-TTM and the language is based on SystemC, opens the possibility to integrate C code provided

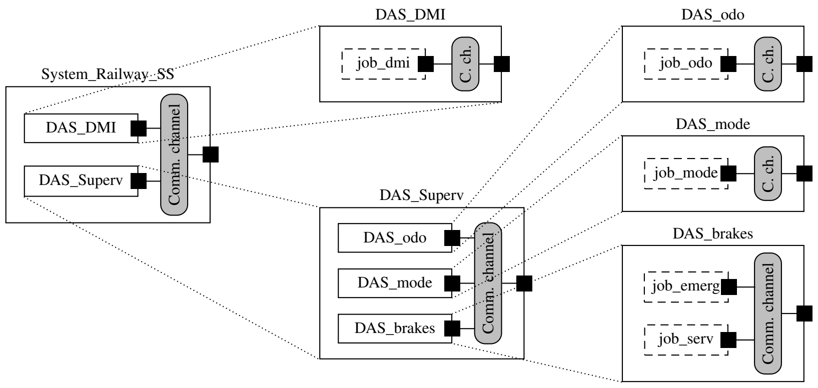


Figure 7.3: Case Study: Platform Independent Model

by different tools and vendors in the PS-TTM model. In this case, we specified the functionality of each job with the graphical Esterel/Ansys SCADE tool. The certifiable C code files generated by the cited tool were then compiled and added to the PS-TTM project as a library. This dramatically reduces the cost of developing new system models and reusing previous models, since the integration of legacy code becomes totally straightforward.

In accordance to the requirements from the standard, we set the period of the system to $250ms$. The frequency of the all the jobs of the SUT is set to 1, so the LET of the jobs is $250ms$.

Reliability assessment of the PIM

This section describes the reliability assessment made to the platform independent model of the simplified railway signaling system. The fault-tolerance provided by the odometry algorithm implemented in the design is evaluated by means of simulated fault injection. To do so, the PS-TTM ATE framework described in Chapter 5 is connected to the SUT and the environment model as Figure 7.4 shows.

As stated before, the model of the environment includes the set of sensors, BTM and TIU. In order to take advantage of the integrability that the PS-TTM provides, the environment has been modeled in Simulink and the C code automatically generated by the Simulink Coder [wwwy] tool has been integrated in the model.

As established by the V-model life cycle usually followed in the development of safety-critical systems, software designers and test engineers work in parallel. The former develop the system by refining components to more detailed abstraction levels, whereas the latter identify the potential failure modes of the systems and create the test cases that will be later used to evaluate the designs. The PS-TTM suggests following the same working scheme, so that, in parallel to the development of the PIM carried out

Listing 7.1: Extract of the PIM code

```
/* das_superv.h */
PITTM_DAS(DAS_Superv){
    ...
    /* Constructor */
    DAS_Superv(const sc_module_name & nm, unsigned int freq)
    : pittm_das(nm, freq)
    {
        /* Declare Components */
        DAS_ODO     *p_das_odo      = new DAS_ODO   ("das_odo", 1);
        DAS_MODE    *p_das_mode     = new DAS_MODE  ("das_mode", 1);
        DAS_BRAKES  *p_das_brakes   = new DAS_BRAKES ("das_brakes", 1);
        /* Place components in hierarchical component */
        v_c.push_back(p_das_odo);
        v_c.push_back(p_das_mode);
        v_c.push_back(p_das_brakes);
        ...
        //Initialize
        initialize(DAS_Superv_Variables_rte, route);
    }
};

/* das_brakes.h */
PITTM_DAS(DAS_BRAKES){
    ...
    /* Constructor */
    DAS_BRAKES const sc_module_name & nm, unsigned int freq)
    : pittm_das(nm, freq)
    {
        /* Declare Components */
        JOB_EMERG   *p_job_emerg= new JOB_EMERG ("emerg_inst", 1);
        JOB_SERV        *p_job_serv = new JOB_SERV ("serv_inst",  1);
        /* Place components in hierarchical component */
        v_c.push_back(p_job_emerg_inst);
        v_c.push_back(p_job_serv_inst);
        ...
        //Initialize
        initialize(DAS_BRAKES_Variables_rte, route);
    }
};

/* job_emerg.h */
PITTM_JOB(JOB_EMERG){
    /* Internal variables */
    tDistanceInt32 s;
    tSpeedInt32 v;
    kcg_bool stdby, superv, reset;
    outC_D21_Decisions_Emergency out;
    /* Constructor */
    JOB_EMERG(const sc_module_name & snm, unsigned int freq)
    : pittm_job(snm, freq){}
    /* Thread */
protected:
    void ctask_thread(void);
    void fcn(void);
};

/* job_emerg.cpp */
void JOB_EMERG::fcn(){...}
void JOB_EMERG::ctask_thread(){
    ...
    while(true)
    {
        /* pull variables */
        s=pull<tDistanceInt32>(e_TTM_COM_IF_LIF, DAS_BRAKES_S);
        ...
        /* compute */
        fcn();
        /* push variables */
        push<kcg_bool>(e_TTM_COM_IF_LIF, DAS_EVC_EMERG, out.Emerg);
        /* wait LET */
        wait();
    }
}
```

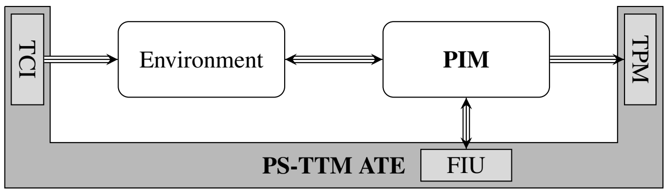


Figure 7.4: Composition of the testing and fault injection environment for the PIM

by the system designers, the test engineers should identify the potential failure modes of the PIM and create its FMEA.

In the present example, the FMEA of the PIM enumerates 109 potential failure modes, rooted in 276 different potential causes. According to the FMEA, the PIM should be able to tolerate 36 potential failure modes, due to the fault-tolerance provided by the selected odometry algorithm, and thus keep the maximum error in the estimation of the distance traveled by the train within the boundaries defined by Equation 7.1. Figure 7.5 shows an extract of the FMEA of the PIM.

The testing team first designs one (or more) significant test cases in a TCI-compliant format and they identify the set of variables to observe, according to the functional properties they want to validate from the model. With this data, they are ready to perform a fault-free simulation of the PIM against the test cases by means of the PI-TTM execution engine and the PS-TTM ATE. The simulation results provided by the TPM of the PS-TTM ATE are stored and analyzed, and once validated, are considered the golden behavior of the system.

Based on the FMEA of the PIM, the test engineers can also design the fault injection campaigns that will assess the robustness of the odometry algorithm against such faults, and simulate them by means of the PS-TTM ATE FIU. Since the golden behavior of the PIM is known, a comparison between the results of the fault-free and faulty simulations of the PIM enables to evaluate the robustness of the system against the faults injected.

In this example, the selected test case simulates a journey of 240 seconds between two stations at a distance of over 7.5 kilometers. Figure 7.6 shows the results recorded by the TPM of the PS-TTM ATE during the fault-free simulation.

Once the fault-free simulation is completed, the reaction of the system against its potential failures identified in the FMEA must be evaluated. To do so, test engineers must design one (or more) fault injection campaigns for each failure mode. The fault campaign is processed by the FIU and applied to the simulation of the PIM. In this case, a total amount of 47 different fault injection campaigns were defined by the testers for the validation of the tolerance of the odometry algorithm against the 36 failure modes identified in the FMEA. Table 7.1 shows an extract of the fault injection campaigns.

Figure 7.7 shows the errors caused by each fault campaign introduced in table 7.1 in comparison with the golden (fault-free) model. The results of the different simulations

Failure Mode and Effects Analysis Worksheet

Process or Product: Simplified ETCS - PIM

FMEA Team:

Team Leader:

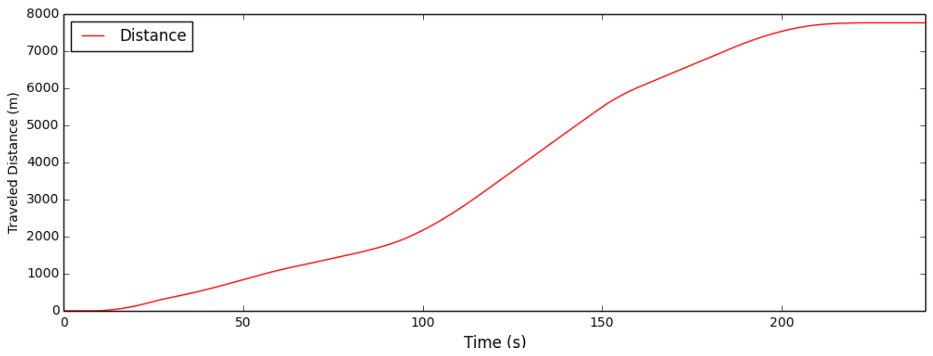
FMEA Number:

FMEA Date: (Original)

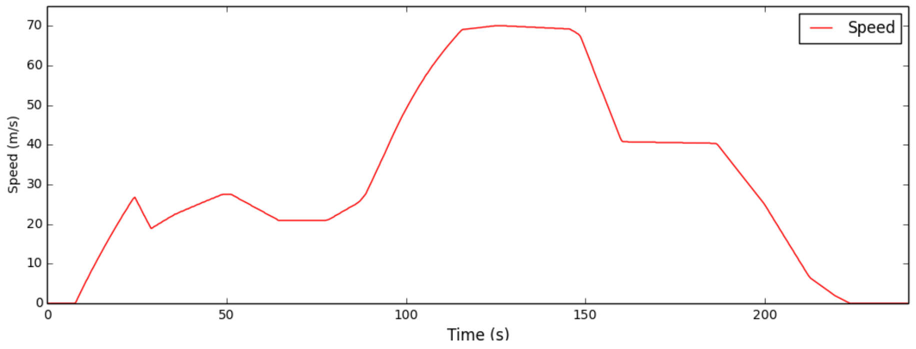
(Revised)

	FMEA Process												Action Results				
Line	Component and Function	Potential Failure Mode	Potential Effect(s) of Failure	Severity	Potential Cause(s) of Failure	Occurrence	Current Controls, Prevention	Current Controls, Detection	Detection	RPN	Recommended Action	Responsibility and Target Completion Date	Action Taken	Severity	Occurrence	Detection	RPN
1	Encoder 1 / Read	Incorrect execution	Erroneous estimation of wheel 1 speed	2	Encoder incorrectly installed	3	Installation manual	--	3	18	Fault Tolerant odometry algorithm						
2					Faulty comm line	4	--	periodic revision	3	24	Fault Tolerant odometry algorithm						
3					Wheel slipping	7	Supervision of wheels / railway	--	3	42	Fault Tolerant odometry algorithm						
4					Wheel blocked (not moving)	7	Supervision of wheels / railway	--	3	42	Fault Tolerant odometry algorithm						
5					Encoder disk broken	4	--	periodic revision	3	24	Fault Tolerant odometry algorithm						
…					…	…	…	…	…	…	…						
10	Encoder 1 / Read	No execution	Erroneous estimation of wheel 1 speed	2	Encoder incorrectly installed	3	Installation manual	--	3	18	Fault Tolerant odometry algorithm						
11					Faulty power supply	4	--	periodic revision	3	24	Fault Tolerant odometry algorithm						
…	…	…	…	…	…	…	…	…	…	…	…						
29	Accelerometer / Read	Incorrect execution	Incorrect estimation of acceleration	3	Accelerometer incorrectly installed	3	Installation manual	--	3	27	Fault Tolerant odometry algorithm						
…	…	…	…	…	…	…	…	…	…	…	…						
40	BTM / Read	No execution	Incorrect distance estimation	3	BTM incorrectly installed	3	Installation manual	--	3	27	Fault Tolerant odometry algorithm						
…	…	…	…	…	…	…	…	…	…	…	…						

Figure 7.5: Extract of the FMEA for the PIM



(a) Estimation of traveled distance

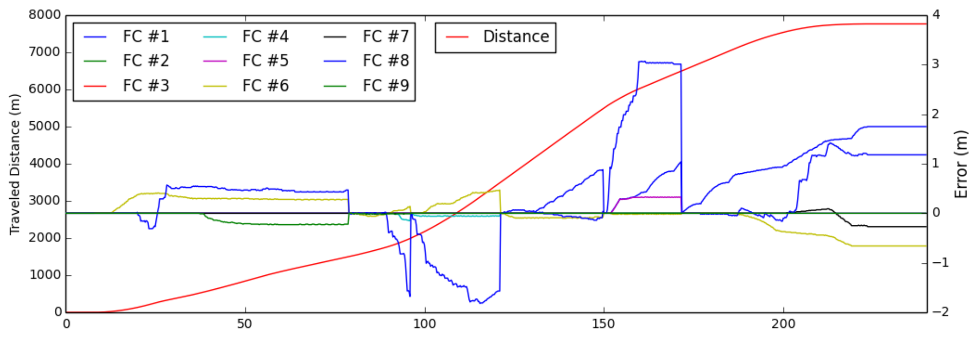


(b) Estimation of speed

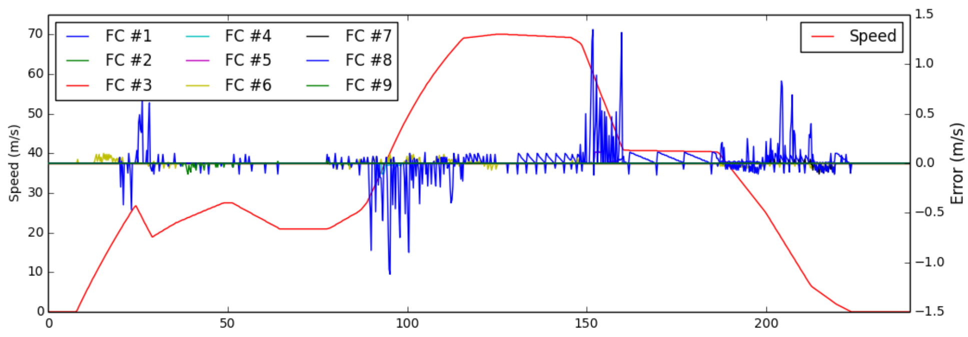
Figure 7.6: Results of the fault-free simulation of the PIM

Table 7.1: Fault injection campaigns for PIM model

#	Fault Location			Fault		Fault Set			Description
	Job	**Entity**	**Type**	**Effect**	**Attributes**	**Mode**	**Trig.time(s)**	**Duration(s)**	
1	job_odo	enc_1	input	I_C	0	p	130.0	–	*Wheel stuck / Encoder broken*
2	job_odo	enc_1	input	I_S	–	p	180.0	–	*Encoder broken (measuring a fix value)*
3	job_odo	enc_1	input	I_R	$0, 600$	p	80.0	–	*Encoder broken (measuring wrong values)*
4	job_odo	enc_1	input	I_C	600	t	91.0	7.0	*Wheel slipping during acceleration*
5	job_odo	enc_2	input	I_C	0	t	150.0	8.0	*Wheel skidding (blocked by brakes)*
6	job_odo	$accel$	input	F_A	1.1	p	0.0	–	*Accelerometer incorrectly installed*
7	job_odo	$accel$	input	F_S	–	p	200.0	–	*Accelerometer broken(measuring a fix value)*
8	job_odo	$accel$	input	F_R	$-2, 2$	p	20.0	–	*Accelerometer broken(measuring wrong values)*
9	job_odo	$accel$	input	F_OR	$-0.1, 0.1$	t	35.0	50.0	*Noise in the signal*



(a) Traveled distance and estimation error due to faults



(b) Speed and estimation error due to faults

Figure 7.7: Estimation errors introduced by fault injection (PIM)

show that odometry algorithm designed for this system provides accurate results in the estimation of the traveled distance even in the presence of faults in the sensors. Overall, the maximum estimation errors occurred during the 8th fault injection campaign, where the maximum error raised up to $3.07m$ when the train had traveled a total distance of $6044.46m$, what leads to an error of 0.05%. This happened $160.5sec$ after the start of the simulation. However, the maximum error in percentage took place during the 6th campaign, and reached an error of 4.76%. Anyway, this happened at instant $8.250sec$, where the traveled distance was still very small ($0.21m$ traveled, $0.22m$ measured due to the fault).

Regarding the estimation of the speed, the experiments made by means of the PS-TTM ATE framework help to strengthen our confidence in the robustness of the algorithm. In fact, the maximum error showed that the fault injection campaigns could introduce in the speed estimation raised up to $1.350m/s$ respect to the non-faulty simulation, at instant $151.75s$ during the 8th campaign, while the train was traveling at $60.230m/s$

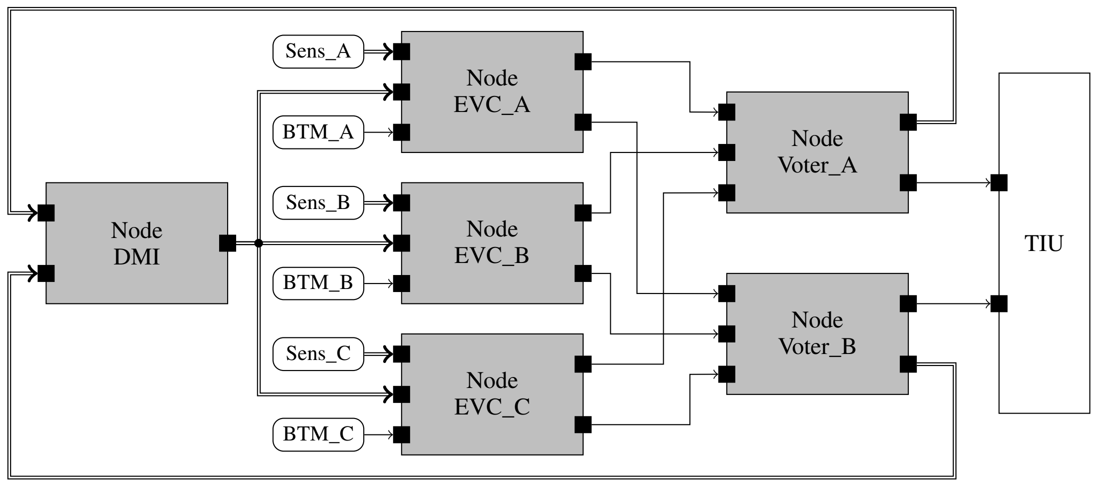


Figure 7.8: Case Study: Functional view emergency brake activation algorithm (PSM)

($61.580 m/s$ where measured due to the fault). This means an estimation error of 2.24%.

All in all, the estimation errors made by the algorithm due to the faults were always smaller than the maximum error rate permitted by the standards (Equation 7.1). As a conclusion, the design and implementation of the odometry algorithm are considered acceptable.

Anyway, the PIM model and the PS-TTM ATE helped the test engineers to identify the special sensitivity that the algorithm showed for faults in the accelerometer. Thus, future work on the algorithm could focus on the improvement of this fact.

Platform Specific Model

Once the functional model (PIM) has been validated, the design is refined by introducing new specifications about the target platform. In this case, the system is built onto a Triple Module Redundant (TMR) architecture, which enables to achieve the required SIL-4 integrity level in accordance to the international EN-50128 safety standard for the railway domain [CEN11]. Redundancy increases the robustness of the system against the failure modes that are not masked by the PIM. Figure 7.8 shows the overall functionality of the PSM of the system. For the sake of simplicity, the figure focuses on the activation system of the emergency brake; however, the *service brake* and the *overspeed warning*, which have been ommited from the figure, would work analogously.

The TMR system is composed of three nodes, each of them hosting a replica of the simplified EVC functionality. Each of the nodes is connected to its dedicated sensors and BTM, so that a fault in the sensors of a node does not have any impact on the rest of the nodes. This increases the dependability level of the system, since it avoids error propagation between replicated nodes, and thus enables to consider each couple of sensors-supervision subsystems a single fault containment region.

As seen in the platform independent model, each replicated node outputs three main signals: *emergency brake*, *service brake* and *overspeed warning*. Since these signals are replicated three times in the system, each of the signals is handled by two exact '2 out of 3' (*2oo3*) voters. Thus, the system contains 6 exact voters altogether.

Each of the voters receives three replicated values from the nodes, and implements an agreement protocol that takes the decision about the activation of the signal. Besides, the voters provide information about the state of the system to the driver via the DMI, by the activation of two warning signals: *single-failure warning* and *system-failure warning*. The former informs the driver that one of the nodes is providing erroneous values, and identifies the corresponding node. The latter warns the driver that the system is suffering from a multiple failure and the train needs to be stopped.

The voters have two operating modes: *normal voting mode* and *degraded voting mode*. The functionality of the voting system is described below:

- The starting operating mode of the voters is *normal voting mode*.
- If the three values received by a voter are equal, the voter remains in *normal voting mode* and forwards the input values to the output value. No warning is sent to the DMI.
- If one of the replicated values received by the voter is distinct to the other two, the voter switches to *degraded voting mode*. In that case the voter behaves as a *1oo2* voter in which the distinct input value is ignored, i.e., the inputs coming from the faulty node are no longer taken into account for the voting algorithm and the result of the *1oo2* algorithm is forwarded to the output. The voter sends a *single failure warning* to the DMI with the identifier of the faulty node.
- If there is a disagreement between the two active inputs when the voter is in *degraded voting mode*, the results of the EVC system are completely ignored and the voter automatically orders the TIU to apply the emergency brakes to the train. In addition, it informs the driver about the multiple failure by sending a *system failure warning* to the DMI.

Having two independent voters for each signal enables to implement different braking systems in parallel, what increases the dependability level of the system. In this case, one of the voters would control the electric braking system of the train whereas the other voter would control the pneumatic braking system. This way, the system keeps the fault-tolerance against a single fault, as a failure in one of the voters would not jeopardize the reliability of the braking system.

Figure 7.9 shows the platform specific model of the system in the PS-TTM. The components in gray are given by the PS-TTM library. The SUT is defined as a cluster

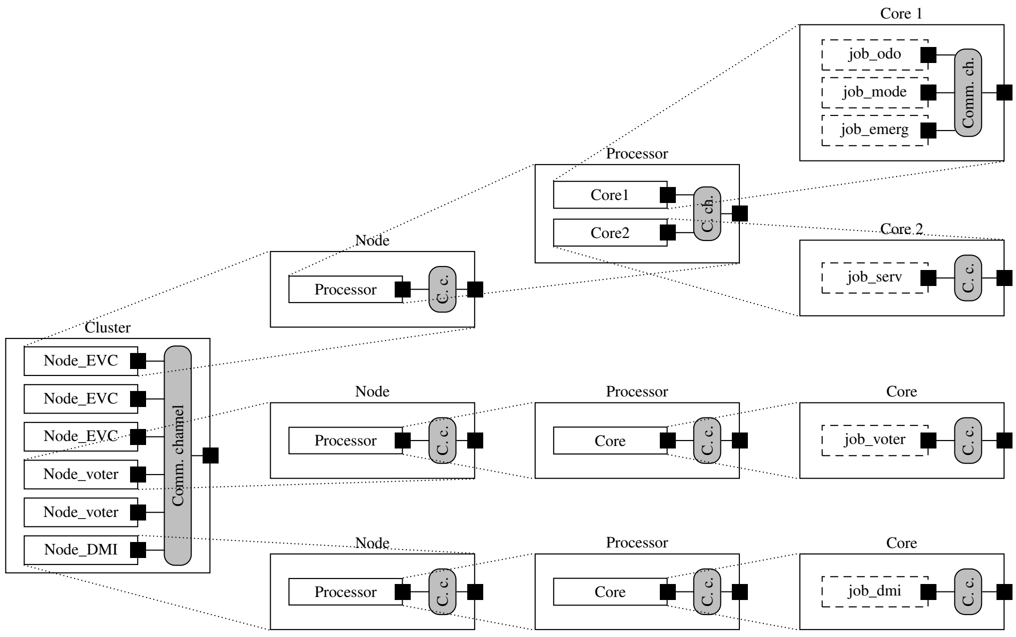


Figure 7.9: Case Study: Platform Specific Model

containing 6 nodes. The redundant simplified EVCs are hosted in three identical nodes containing a dual-core processor. The first of the cores is the host for the safety-critical jobs, i.e., the odometry job, the mode control job and the emergency brake control job, whereas the second core hosts the service brake control job. Two more nodes are designed for the voters, which are composed by a single core processor running the voting algorithm described below. The last voter is dedicated to the DMI, and also contains a single-core processor.

We design the functions with SCADE and we generate C code automatically using the KCG tool. C code for the voters is also automatically generated from SCADE models.

Listing 7.2 shows an extract of the textual representation of Figure 7.9 following the syntax of the PS-TTM, where the general structure of the PSM can be observed. As in the case of PIMs, jobs are considered atomic components in platform specific models. Hence, at PSM level the designers are still open to integrate C code provided by different tools in their model. In this case, we again define the functionality of jobs by integrating the compiled C code files generated with the KCG code generator of the SCADE tool as a library in the PS-TTM project. Due to the lighter complexity of the voting algorithm in comparison to the odometry algorithm and simplified EVC functionality, we decided to implement the functionality of the voters typing directly the C code by hand.

As Listing 7.2 shows, in this case the designer decided to create a new constructor

Listing 7.2: Extract of the PSM code

```
/* cluster_superv.h */
PSTTM_CLUSTER(Superv)
{
    ...
    /* Constructor */
    Superv ( const sc_module_name & sname) : psttm_cluster(sname)
    {
        /* Declare Components */
        NODE_DMI        *p_dmi_inst     = new NODE_DMI   ("node_dmi"  );
        NODE_EVCA       *p_evc_instA    = new NODE_EVCA  ("node_evc_A");
        NODE_EVCB       *p_evc_instB    = new NODE_EVCB  ("node_evc_B");
        NODE_EVCC       *p_evc_instC    = new NODE_EVCC  ("node_evc_C");
        NODE_VOTERA     *p_voter_instA  = new NODE_VOTERA("node_vot_A");
        NODE_VOTERB     *p_voter_instB  = new NODE_VOTERB("node_vot_B");
        /* Place components in hierarchical component */
        v_c.push_back(p_dmi_inst);
        v_c.push_back(p_evc_instA);
        ...
        //Initialize
        initialize(Superv_Variables_rte,  route);
    }
};

/* processor_node_evc_A.h */
PSTTM_PROCESSOR(PROC_EVCA)
{
    ...
    /* Constructor */
    PROC_EVCA ( const sc_module_name & sname) : psttm_processor(sname)
    {
        sc_time period(250, SC_MS);
        /* Declare Components */
        CORE_1_EVCA *p_core_a_evc_inst      = new CORE_1_EVCA("core_A", period);
        CORE_2_EVCA *p_core_b_evc_inst      = new CORE_2_EVCA("core_B", period);
        /* Place components in hierarchical component */
        v_c.push_back(p_core_a_evc_inst);
        v_c.push_back(p_core_b_evc_inst);
        //Initialize
        initialize(ProcEVC_A_Variables_rte, route);
    }
};

/* core1_evc_A.h */
PSTTM_CORE(CORE_1_EVCA)
{
    ...
    /* Constructor */
    CORE_1_EVCA ( const sc_module_name & sname, ttm_dt_time period) : psttm_core(sname)
    {
        /* Declare Components */
        JOB_ODOA    *p_job_odoA_inst    = new JOB_ODOA  ("job_odoA");
        JOB_MODEA   *p_job_modeA_inst   = new JOB_MODEA ("job_modeA");
        JOB_EMERGA  *p_job_emergA_inst  = new JOB_EMERGA("job_emergA");
        /* Place components in hierarchical component */
        v_c.push_back(p_job_odoA_inst);
        v_c.push_back(p_job_modeA_inst);
        v_c.push_back(p_job_emergA_inst);
        /* Set temporal specifications of each job */
        ttm_dt_time_cycle stime_descr(period);
        stime_descr.add(p_job_odoA_inst, 0.0);
        stime_descr.add(p_job_modeA_inst, 0.0);
        stime_descr.add(p_job_emergA_inst, 0.0);
        //Initialize
        initialize(stime_descr, Core1_EVCA_Variables_rte, route);
    }
};
```

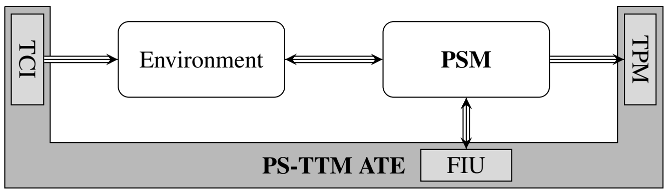


Figure 7.10: Composition of the testing and fault injection environment for the PSM

for the *core* components of the model. This constructor contains a `period` parameter besides the `sname`, and it is used by the *core* to define the temporal properties of each job running in it. The temporal specifications of each job are defined in the `stime_descr` vector, which contains a period plus a phase delay specification for each job.

In this case study, the period of all the cores of in the system are set to $250ms$, and a phase delay of 0.0% is defined for all the jobs, so that the timing properties defined in the PIM remain unchanged in the PSM.

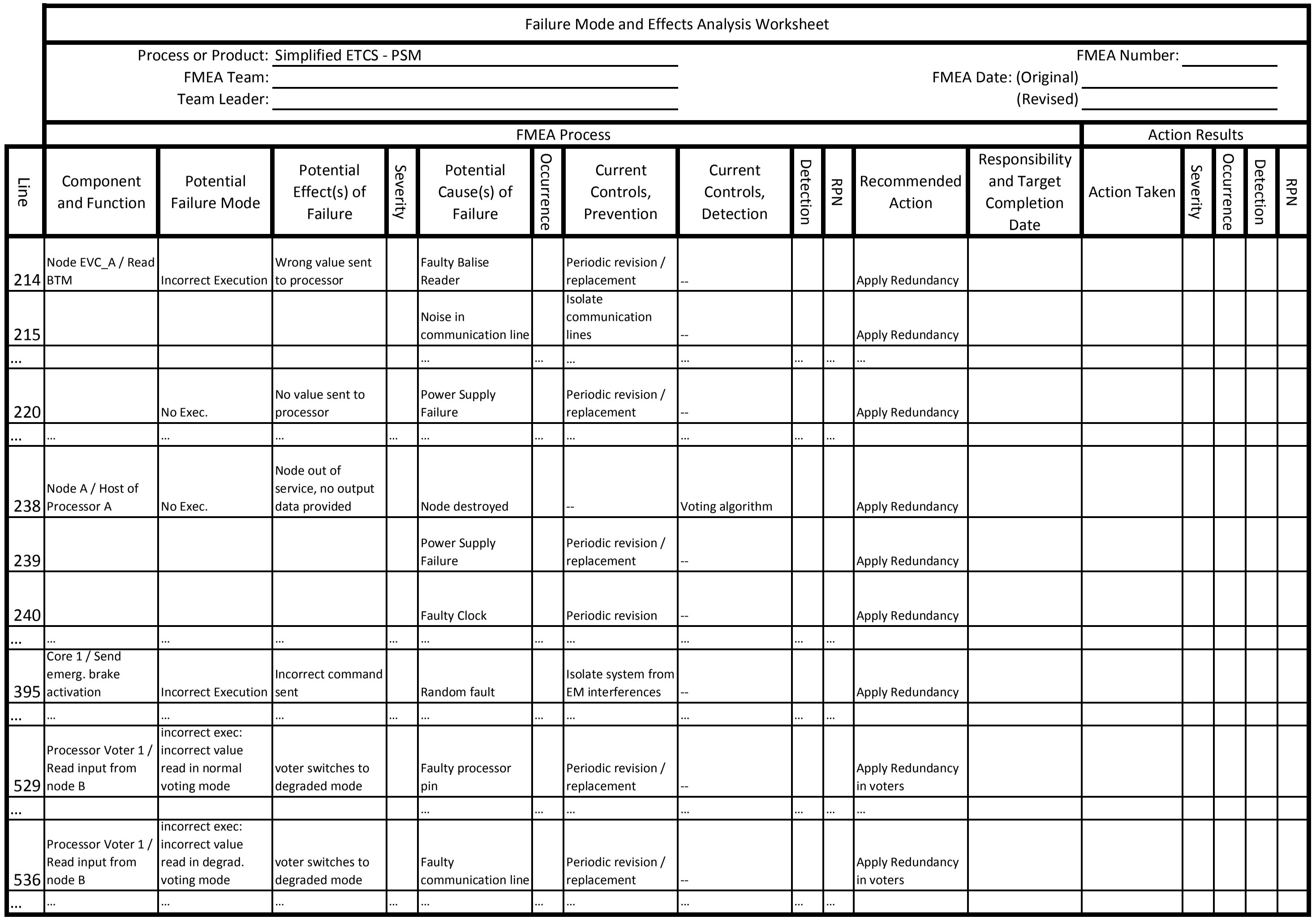
Reliability Assessment of the PSM

Once the PSM of the simplified railway signaling system has been modeled by means of the PS-TTM, its evaluation its performed by simulation and fault injection. In this case, we assess the fault-tolerance mechanisms introduced in the platform specific model of the system, primarily focusing on the evaluation of the behavior of the voters.

To do so, the PS-TTM ATE is again connected to the SUT and the environment model as shown in Figure 7.10. In this case, the model of the environment includes the components in white from Figure 7.8, i.e., the set of sensors and BTM of each replicated node, and the Train Interface Unit. This environmental components have been modeled with Simulink, and the C code automatically generated with Simulink Coder [wwwy] has been integrated in the model, taking advantage of the integrability offered by the PS-TTM.

The FMEA developed in the current example enumerates a total of 282 potential failure modes, rooted in 608 potential causes. According to the specifications, the system should mask 261 of those potential failure modes (92%) by the introduction of triple modular redundancy in the supervision system and double redundancy in the voters. The remaining 21 failure modes that are not tolerated by the PSM are the ones caused by systematic errors in the software design. In order to gain tolerance against such faults, the system would need to apply software diversity in each replicated node. Figure 7.11 shows an extract of the FMEA of the PSM.

The assessment of the platform specific model follows the same work-flow followed during the evaluation of the PIM. The testing team first designs one (or more) significant



Failure Mode and Effects Analysis Worksheet

Process or Product: Simplified ETCS - PSM

FMEA Team:

Team Leader:

FMEA Number:

FMEA Date: (Original)

(Revised)

	FMEA Process												Action Results				
Line	Component and Function	Potential Failure Mode	Potential Effect(s) of Failure	Severity	Potential Cause(s) of Failure	Occurrence	Current Controls, Prevention	Current Controls, Detection	Detection	RPN	Recommended Action	Responsibility and Target Completion Date	Action Taken	Severity	Occurrence	Detection	RPN
214	Node EVC_A / Read BTM	Incorrect Execution	Wrong value sent to processor		Faulty Balise Reader		Periodic revision / replacement	--			Apply Redundancy						
215					Noise in communication line		Isolate communication lines	--			Apply Redundancy						
…					…	…	…	…	…	…	…						
220		No Exec.	No value sent to processor		Power Supply Failure		Periodic revision / replacement	--			Apply Redundancy						
…	…	…	…	…	…	…	…	…	…	…							
238	Node A / Host of Processor A	No Exec.	Node out of service, no output data provided		Node destroyed		--	Voting algorithm			Apply Redundancy						
239					Power Supply Failure		Periodic revision / replacement	--			Apply Redundancy						
240					Faulty Clock		Periodic revision	--			Apply Redundancy						
…	…	…	…	…	…	…	…	…	…	…							
395	Core 1 / Send emerg. brake activation	Incorrect Execution	Incorrect command sent		Random fault		Isolate system from EM interferences	--			Apply Redundancy						
…	…	…	…	…	…	…	…	…	…	…							
529	Processor Voter 1 / Read input from node B	incorrect exec: incorrect value read in normal voting mode	voter switches to degraded mode		Faulty processor pin		Periodic revision / replacement	--			Apply Redundancy in voters						
…					…	…	…	…	…	…	…						
536	Processor Voter 1 / Read input from node B	incorrect exec: incorrect value read in degrad. voting mode	voter switches to degraded mode		Faulty communication line		Periodic revision / replacement	--			Apply Redundancy in voters						
…	…	…	…	…	…	…	…	…	…	…							

Figure 7.11: Extract of the FMEA for the PSM

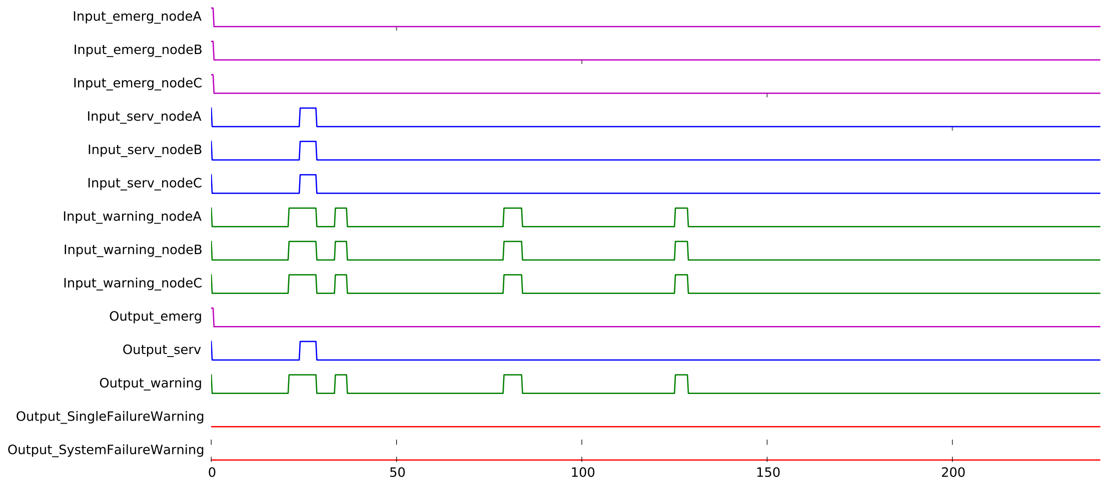


Figure 7.12: Results of the fault-free simulation of the PSM

test cases in a TCI-compliant format and they identify the set of variables to observe, according to the functional properties they want to validate from the model. With this data, the PS-TTM ATE can carry out a fault-free simulation of the PSM for the specified test cases.

In this case, the assessment of the PSM is performed with the same test cases used for the validation of the PIM. As mentioned before, the validation of the PSM will mainly focus on the evaluation of the behavior of the redundant supervision subsystems and the voters. Therefore, the test-point configuration file specifies the outputs of these subsystems, which will be tracked and recorded by the TPM for a later analysis.

The fault-free simulation campaign provided the results shown in Figure 7.12 for the mentioned test case. During the fault-free simulation of this test case, the system activated the overspeed warning 4 times, and the service brake once. As expected, the failure and system-failure warnings were not activated.

Once the fault-free simulation is completed, we continue the assessment of the PSM with the evaluation of its behavior against the potential failures identified in the FMEA. With this purpose, the test engineers must design the corresponding fault injection campaigns for each failure mode. The fault campaign is processed by the FIU during the initialization phase of the simulation and applied to the model during the its execution. In the case of the PSM, a total of 233 fault injection campaigns were defined by the testers for the validation of the tolerance properties introduced by the redundancy and voting algorithms of the PSM. Table 7.2 shows a brief extract of the fault injection campaigns.

Table 7.2: Fault injection campaigns for PSM model

#	Fault Location	Fault		Fault Set			Description
		Effect	Attributes	Mode	Trig.time(s)	Duration(s)	
1	`Node_EVC_A`	`NE`	–	p	85.0	–	*Node A stopped working*
2	`Proc_EVC_B`	`C`	–	p	20.0	–	*Processor B provides incorrect results*
3	`EVC_C_Core1`	`OoT`	0.50	p	120.0	–	*Core 1 of a processor C is out of time bounds*
4	`Node_EVC_A`	`B`	–	t	40.0	25.0	*Node A babbling, incorrect results*
5	`Node_EVC_B, Proc_EVC_C`	`NE, C`	–	p, p	60.0, 150.0	–	*Double failure (Node B stops, then processor C incorrect)*
6	`job_serv_A` $serv$ *output*	`B_I`	–	t	160.0	0.50	*Bit-flip in Service. Brake signal sent by node A*
7	`job_emrg_C` $emrg$ *output*	`B_OC`	–	t	105.0	15.0	*Emerg. Brake not received from Node C (noise)*

Table 7.3: Temporal properties of failure detection in fault injection campaigns (PSM)

#	Fault trigger instant (s)	Fault warning activ. instant (s)	System fault activ. instant (s)
1	85.0	125.25	-
2	20.0	20.25	-
3	120.0	125.25	-
4	40.0	40.25	-
5	60.0, 150.0	79.00	150.25
6	160.0	160.25	-
7	105.0	105.25	-

The simulation results show that all the faults injected in the system during the different campaigns were effectively detected by the voters. The functionality of the voters also provided successful results, since the appropriate warning messages were sent to the DMI and the voting algorithm forwarded the expected results to the TIU. Table 7.3 summarizes the temporal characterization of the detection of failures in the system by the voters.

As the table shows, all the faults injected in the system during the simulations were detected by the voters. Since we configured all jobs in the system with a period of one macrotick (250ms), *corruption* and *babbling* faults were detected 250ms after their injection in the system, as expected. *Bit-flips* in signals and *open circuits* were also detected in the next macrotick.

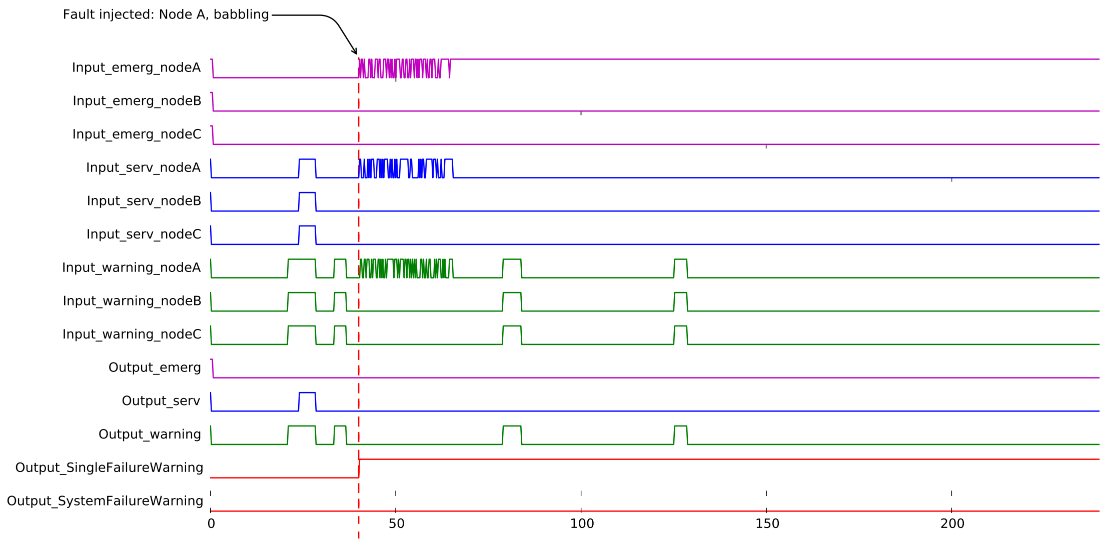
However, *no-execution* and *out of time* faults, injected in the 1st, 3rd and 5th fault configurations, took longer to detect. This happened because, due to the state of the system at the moment of the injection, the faults were dormant. In fact, *no-execution* and *out of time* faults do not become active until the value of the signal changes, since they do not cause an alteration of the signal values by themselves.

As an example, Figure 7.13 shows the results recorded by the TPM of the PS-TTM ATE for the simulations of the fault configurations n^{o} 4 and 5. The 4th fault configuration establishes to inject a *babbling* fault to supervision node A during 25.0 seconds, starting at instant 40.0. Figure 7.13a shows the evolution of the inputs and outputs of the voter A during the simulation of the 4th FC. We can observe in the figure how, during 25 seconds starting from the 40th, the voter receives random values from the signals coming from the supervision node A, due to the signal sabotaging performed by the FIU of the PS-TTM ATE. When the voter receives a different value from the signals of node A, it reacts by activating its *single failure warning* as specified in the requirements. In addition, it ignores the the signals coming from the mentioned node, so that it calculates its output values by performing a *1oo2* algorithm between the signals from node B and C. Thus, since there is no disagreement between the values provided by that nodes from that instant, the voter simply forwards the values received from both active nodes to the TIU.

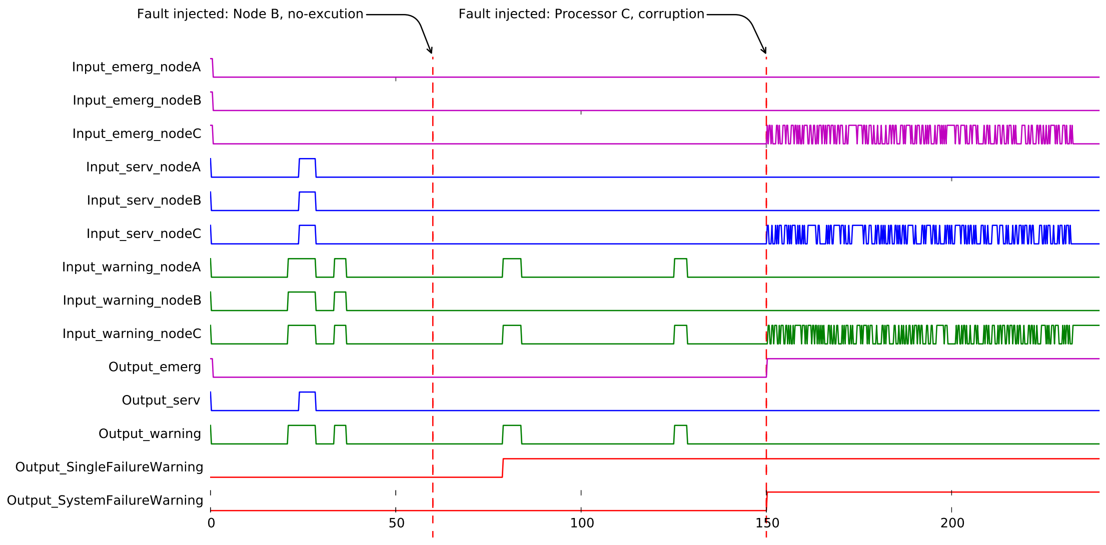
Regarding to the fault campaign n^{o} 5, the specification states that two faults must be injected in the system. The first of them is a permanent *no-execution* fault in the supervision node B at instant $60.0s$. The second one is a *corruption* fault in the processor of the supervision node C at instant $150.0s$, and will also have a permanent character. According to the requirement specifications, this type of multiple failures should make the system halt and automatically activate the emergency brakes to the system.

Figure 7.13b shows the response of the voter to such a fault campaign. We can observe in the figure how, although the fault is injected 60.0 seconds after the start of the simulation (in simulation time), the voters detect the failure at instant $79.0s$. As mentioned before, this occurs due to the dormant state of the injected fault. In other words, since the FIU injects a *no-execution* fault permanent, the output signals of the faulty node do not change the value anymore in the simulation. However, since due to the test case specification, the correct nodes do neither change their output values, the voters are unable to detect the failure. The detection however takes place at the instant $79.0s$, $250ms$ after the moment in which the nodes A and C command the activation of the overspeed warning signal. Node B does not command the activation due to the *no-execution* fault injected previously. This enables the voter to detect the faulty behavior of the node, what makes it to set the *single failure warning* and switch to the degraded voting mode.

The second fault injected during FC 5 caused a disagreement between the two active supervision nodes (A and C). Thus, as specified by the fail-safe characterization of the system, the voter ignored all the signals coming from the nodes and commanded the TIU to apply the emergency brakes in order to stop the train safely and avoid any risk.



(a) Simulation results of test case with fault campaign 4



(b) Simulation results of test case with fault campaign 5

Figure 7.13: Results of simulations with different fault campaigns

In addition, it sent a *system failure warning* to the DMI to inform the driver about the multiple failure detected in the system.

In conclusion, the successful response of the PSM to all the fault configurations defined by the test engineers according to the FMEA allows us to state that the design of the application as a TMR system with double redundant voters, and the implementation of the supervision system and voting algorithm comply with the requirements and are therefore considered acceptable. However, the experiments made by the PS-TTM ATE identify a drawback of the design regarding the incapacity of the system to detect dormant faults. To overcome this drawback we suggest to include timestamps in the data sent by the redundant supervision subsystems. This way, the voters would detect a disagreement between the timestamps provided by the redundant nodes with a delay of just one macrotick, which would guarantee to identify the failure and thus would enable to apply the required fault-tolerance mechanisms at the next macrotick.

7.3 Results and Discussion

The experience of the case study developed for the evaluation of the PS-TTM allows us to extract some interesting conclusions, which are summarized in this section.

First, the PS-TTM framework enabled to straightforwardly model the intended system at the PIM and PSM levels of abstraction. The automatic handling of the temporal properties of communications and triggering instants of jobs relaxed the complexity of designing the system in SystemC code, since it prevented us from having to calculate and explicitly specify the mentioned parameters for each job and communication channel. This way, we could focus on the design of the functionality of the system. In fact, thanks to the PS-TTM, the most time-consuming tasks during the case study were the gathering of the requirements and the development of the FMEAs, which fall out of the scope of this thesis.

The time determinism provided by the simulator was validated by manually checking the temporal properties of communications, and has been confirmed as expected. The time-deterministic simulation environment enabled to evaluate both the PIM and PSMs of the system successfully. In this case, the designed test case simulated a journey of 240 seconds between two stations at a distance of over 7.5 kilometers. The simulations were executed using a quad-core laptop running at 2.60 GHz, under a Windows 7 SP1 operating system with SystemC 2.3.0. The simulations of the PIM took between 1.01 and 1.18 seconds, depending on the faults injected and the load of the non real-time operating system. Similarly, the different executions of the PSM needed between 6.19 and 6.72 seconds to run completely.

Thus, in the case of the platform independent model, simulations showed a relationship of at least x200 between simulated time and simulation time. This relationship reached a value of x35 in the case of the platform specific model. Obviously, these results are strongly dependent on the complexity of the models designed with the PS-TTM and might improve or worsen depending on the specific model. All in all, as a first case study, we consider the simulation-speed results very promising.

The testing and SFI capabilities of the PS-TTM ATE have been validated by means of the case study. Its ability to inject faults non-intrusively has demonstrated to be a major advantage, since it has enabled to validate a single model against a considerable number of faults straightforwardly. The effectiveness and usefulness of the SFI has been demonstrated since it enabled to identify the main weaknesses of the railway system we designed, which were unknown until the analysis of the simulation results was carried out.

Namely, we were able to identify two main weaknesses in our railway-system design. First, the odometry algorithm showed an excessive sensitivity against faults in the accelerometer sensor, which was not expected. Although the design of the algorithm was considered successful, the future work should focus on improving its tolerance against such faults. In this case, the repeatability of test campaigns and fault injection activities offered by the PS-TTM and PS-TTM ATE will enable to effectively check the theoretical improvement implemented in the next version of the odometry algorithm. It must be noted that the strengths and weaknesses of the odometry algorithm could be identified in the PIM of the system, long before an operational prototype was built. This represents a big advance in the sense of early verification and validation, which contributes to avoid the propagation of design bugs throughout the development process.

Second, the analysis of the simulation results provided by the PS-TTM ATE for the PSM of the railway system enabled to identify another drawback of the design: *No-execution* and *out of time* faults, injected in the 1st, 3rd and 5th fault configurations, could not be detected by the voters in the next macrotick to their occurrence, but took longer to detect. This happened because, due to the state of the system at the moment of the injection, the faults were dormant. In fact, *no-execution* and *out of time* faults do not become active until the value of the signal changes, since they do not cause an alteration of the signal values by themselves. The design of the system was considered successful anyway, since this fact does not affect the functionality of the system. However, we consider adding timestamps to the messages in the following versions of the railway system, in order to improve its fault-detection capability.

Compared to other modeling approaches, the PS-TTM has shown to provide some promising capabilities. In the following, we briefly compare the PS-TTM to the approaches identified in the State of the Art (Section 3) for the case of our case study.

UML, the most widely used modeling approach for software development, misses the notion of time in its diagrams, which hinders the development of time-triggered software. Moreover, as a software oriented language, it does not directly support the design of HW components, and it is therefore not suitable to model HW/SW systems.

SysML is the adaptation of UML for the development of systems including combinations of HW, SW, data, people, facilities and objects. However, it suffers from the same drawback as UML, i.e., it misses the notion of time in its diagrams, which complicates the generation of executable models from SysML designs.

MARTE, as the UML profile for the modeling of real-time embedded systems, enables to establish temporal properties of systems by allowing to reference time instants from the components of the system. MARTE provides both basic and advanced time modeling concepts, and it is not restricted to a specific MoC in order to enable the users to define their own MoC. However, precisely this fact hinders the development of time-triggered dependable systems without first developing a user-defined time-triggered profile for MARTE which will guarantee the timing coherence of the systems.

For example, the design of the simplified ETCS version of our case study in MARTE would have required the creation of a discretized clock, and the designer should have ensured that all its functional components are related to that clock. Then, he should have identified alternating slots of activity and silence in the timeline in order to build a sparse model of time, and he should have ensured that all the components of the system execute and communicate with each other in compliance with that sparse-time model. Any non-time-triggered component, such as event-triggered tasks or logical clocks should be avoided from the design by the designers themselves. Even in that case, certifying such a system would be a difficult task, since the modeling architecture does not guarantee that models adhere to the time-triggered paradigm. Thus, certification authorities could require designers to formally prove the adherence of the model to the time-triggered MoC, or at least to prove the temporal predictability of their models.

In contrast, modeling the simplified ETCS with PS-TTM has demonstrated to be straightforward. In fact, the creation of the platform independent model only required to define a frequency for each job of the system, thus leaving the management of communications, triggering instants and delay times to the simulation engine. This way, the adherence of the model to the LET MoC was given by the modeling architecture itself. In a second stage, the PIM was transformed into a PSM. In this case, the users were required to provide more detailed timing information explicitly; however, the timing constraints of the PSM could be straightforwardly calculated from the previous PIM. All in all, non time-triggered components, such as event-triggered jobs or logical clocks are not considered by the PS-TTM, which forces engineers to fit all activities into a time-triggered design thus improving the predictability of the systems and facilitating their certification.

Moreover, the separation of concerns in different diagrams provided by UML and its profiles like MARTE complicates to ensure the consistency of the properties throughout the different diagrams that make up a model. The single-description approach of PS-TTM avoids inconsistencies, thus increasing the maintainability of the models.

Similarly to MARTE, AADL enables the designers to define the temporal properties of their systems, and supports the modeling of both SW and HW components. Execution time of threads and frequencies of clocks can be established by the definition of properties of components. However, like in the case of MARTE, AADL itself defines no specific model of computation. Thus, in the case of time-triggered systems, it is the responsibility of the designer to define the system in such a way that the desired temporal constraints are guaranteed, and to demonstrate so to the certification authorities if required.

In summary, the lack of native underlying MoCs in MARTE and AADL hinders the development of time-deterministic models and appropriate simulators for them. Furthermore, the Automatic Test Executor included in PS-TTM enables developers to straightforwardly evaluate the reaction of their models and fault-tolerance mechanisms to faults. Performing such an activity with MARTE or AADL would require the designers to develop new models including faulty components, which would extend the time needed for dependability assessment, thus increasing validation and verification costs.

On the other hand, similarly to PS-TTM, the Distributed Application Layer (DAL) also enables to model the system in terms of SW and HW components, and enables to map SW components to HW ones. The DAL modeling environment is focused on the development of streaming systems, where transmission speed in communications is preferred over dependability. Therefore, the underlying MoC for DAL models is the KPN MoC, which is not a time-triggered MoC. In fact, the KPN MoC does not directly address the timing properties of the communications, but only their partial chronological order. Thus, the KPN MoC is considered an untimed MoC and it is not considered appropriate for safety-critical real-time systems.

Out of all the modeling environments proposed in the state of the art, the FTOS is the closest to the PS-TTM. FTOS provides a modeling framework for dependable systems based on the LET MoC, and focuses on the development of appropriate fault-tolerance mechanisms. However, the modeling work-flow proposed by FTOS (Figure 3.2) forces the designers to start the development of the system with the definition of a hardware architecture model, and to continue with the SW architecture model after the former has been finished. This fact complicates the early validation of some fault-tolerance properties, since it requires to model both the HW and the SW before any test can be run. For instance, in the case of our case study, the detection of the accelerometer fault dependency drawback of the odometry algorithm would have only been possible at the latest stage of the design, whereas the PS-TTM enabled to identify it at the initial platform independent model.

Finally, the E-TTM is focused on modeling the functionality of TTA-based systems in SystemC. Hence, it does not provide any technique to model HW-related components, so it does not support the definition of platform specific models by itself. Besides, it does not provide any native mechanism to perform fault injection non-intrusively, which complicates the verification of the fault-tolerance properties of the models. The PS-TTM takes the core concept developed in the E-TTM and extends it in order to support the definition of PSMs, and enables non-intrusive simulated fault injection for validation and verification of the models.

CHAPTER 8

Conclusion

The present chapter reviews the Platform Specific Time-Triggered Model (PS-TTM) modeling approach and the PS-TTM ATE testing and simulation framework, analyzes the contribution of the work and provides possible future paths to continue with the research.

8.1 Review

This thesis describes PS-TTM, a novel modeling framework for time-triggered safety-critical systems, and PS-TTM ATE, a testing and simulated fault injection framework for the assessment of the fault-tolerance properties of such systems.

The PS-TTM provides a time and value deterministic modeling and simulation framework that enables to design and exercise time-triggered safety-critical systems. The approach proposed in this work is based on the well-known MDA and Y-chart development processes. The approach suggests to separate the specification of the functionality of the system from the description of the target platform in a first design phase. This way, the design of the system begins with the definition of its Platform Independent Model (PIM), which enables designers to focus on a single aspect of the design at the beginning (functionality). Thus, the design of the systems at first glance is eased and the complexity of the models is reduced. Once the functional model has been fully defined, the notion about the platform is added with the generation of the Platform Specific Model (PSM). This model, though more complex than the PIM, enables the designers to focus on other properties of the system, such as their fault-tolerance level against HW faults. Besides, the separation of concerns between functional and platform-related aspects increases the portability and re-usability of the designs.

The complexity challenge is faced in different ways by the PS-TTM. First, as just mentioned, it relies on the separation of concerns as the main characteristic of the development process. Second, instead of defining a new modeling language for the design of the systems, the PS-TTM has been built as a library that extends the well-known SystemC language with a set of functions, structures and macros that enable to define time-triggered safety-critical systems in a straightforward manner. Third, the PS-TTM infrastructure provides a global notion of time for all the components within a model, which is based on the sparse concept of time, as defined by the Time-Triggered Architecture (TTA). And fourth, the PS-TTM enables users to apply abstraction, partitioning and segmentation to the models, at both PIM and PSM level.

On the other hand, the PS-TTM ATE is a testing and simulated fault injection framework for the verification and validation of PS-TTM based systems. The PS-TTM ATE makes possible to simulate both PIM and PSM by means of the PI-TTM and PS-TTM simulation engines respectively. This fact enables the designers to detect design pitfalls at the very early stages of the development of the system, thus reducing the chance to discover bugs late in the design, which could require an expensive system re-design and possibly re-implementing.

The non-intrusive fault injection technique implemented in the PS-TTM ATE enables testers to inject faults in the system during simulation without the need to modify it. The non-intrusiveness makes it possible to inject several faults in a given model without re-compiling it every time a new fault has to be injected. This provides two major benefits: First, since the injection of the fault does not require any modification in the models, there is no chance to modify its fault-tolerance properties due to the fault injection itself, in contrast to intrusive techniques. Second, since the fault injection does not require to modify the model, the latter does not need to be re-compiled every time a fault has to be injected. This means that a given compiled model can be tested against a number of faults, and thus, time consuming compilations only need to be carried out when the designers decide to modify the model by themselves.

Both the PS-TTM and the PS-TTM ATE have been assessed by means of a case-study based on a simplified version of the European Train Control System (ETCS). The case study showed how the PS-TTM could be used to model a complex system with diverse fault-tolerance mechanisms at different abstraction levels. The models were validated by means of the PS-TTM ATE, which provided some promising results, since it permitted us to assess the correctness of the odometry algorithm implemented in the system and the functionality of the *2oo3* voters. More interestingly, the simulations performed by the PS-TTM ATE enabled us to identify the main drawbacks of the design, such as the sensitiveness of the odometry algorithm to faults in the acceleration sensor and the inability of the voters to detect dormant faults. The simulation speeds of the different experiments were also very promising, since they were able run simulations about 200 times faster than the simulated time for PIMs, and 35 times faster than the simulated time for PSMs.

8.2 Analysis of the Contribution

This section summarizes the main contributions provided by this research work:

- *Time-triggered safety-critical systems modeling framework*: The PS-TTM is a modeling framework for the development of time-triggered safety-critical systems, which relies on the 'separation of concerns' concept introduced by the MDA and the Y-chart approach. This separation of concerns relaxes the intrinsic complexity of the design of safety-critical systems. Besides, the designers might also make use of abstraction, partitioning and segmentation techniques in their models in order to ease the design. The definition of models takes SystemC as its underlying design language.

- *Time and value deterministic simulation*: The PS-TTM provides a time and value deterministic framework for the simulation of the models, based on two simulation engines: The PI-TTM is a novel simulation engine that extends SystemC with the temporal constraints introduced by the Logical Execution Time (LET) MoC, and enables to execute LET-based Platform Independent Models (PIMs). The second simulation engine, called PS-TTM, is used for the simulation of Platform Specific Models, and relies on the TT MoC introduced by E-TTM.

- *Mixed abstraction level simulation*: The seamless connection between the PI-TTM and PS-TTM simulation engines makes it possible to simulate systems whose components are modeled at different abstraction levels. This can be a major advantage, since, although different parts of a given system are typically modeled in parallel, the working pace is not always the same. Therefore, it is not unusual to reach the situation in which one part of the system is at a PSM level, whereas another part is still at a PIM level. In such a situation, the PS-TTM would still allow the test engineers to validate one of the subsystem by simulating the whole system making use of the mixed abstraction level simulation capability.

- *Testing and Simulated Fault Injection Framework*: The PS-TTM Automatic Test Executor (ATE), is a testing and SFI framework for the verification and validation of PS-TTM compliant models. The PS-TTM ATE enables to exercise both PIMs and PSMs with the test cases specified by the test-engineers, and provides a high level of observability within the models. The simulated fault injection technique is non-intrusive, i.e., it does not require to modify the models to get the faults injected. Besides, it provides a single specification language (based in XML) for the definition of test cases, test points and fault injection campaigns, which enables to reuse the test activities through the different stages of the development process, from PIM to PSM.

- *Fault libraries for PIMs and PSMs*: The PS-TTM ATE includes fault libraries for both PIMs and PSMs. This way, the test engineers can invoke a fault by just specifying the name of the desired fault effect in the fault-configuration file, which facilitates the definition of the fault campaigns and increases the usability of the PS-TTM ATE. The fault library for PIMs focuses in faults at signal levels, whereas the fault library designed specifically for PSMs provides a set of more abstract HW-related faults.

- *Heterogeneous systems simulation*: The fact that the PS-TTM modeling framework has been built as an extension to the SystemC language enables to integrate PS-TTM components at PIM or PSM level with other components that might rely on different MoCs, such as continuous subsystems (simulated by means of SystemC-AMS), physical subsystems (simulated with MVDP), or synchronous subsystems (simulated directly with SystemC). This enables engineers to include PS-TTM subsystems as components of cyber-physical systems, embedded systems that interact with sub-components of very different computing and physical properties, which are getting an increasing interest nowadays.

8.3 Future Work

The research work presented in this dissertation could be further extended in diverse ways, such as the following:

- *Automatic code generation from MARTE diagrams*: As mentioned in the state of the art of this dissertation, the MARTE profile for UML enables to capture the temporal properties of UML blocks directly in the diagrams, which eases the specification of the design of HW/SW systems. Although the MARTE profile does not limit its application to time-triggered systems, we could still provide an automatic PS-TTM code generator tool for the subset of time-triggered components from MARTE. Thus, we could benefit from the contributions provided by the PS-TTM and PS-TTM ATE with the advantage of supporting the already well-known UML / MARTE modeling standard.

- *Integration with "virtual platform simulators"*: Virtual platform simulators permit to accurately simulate HW/SW systems, by exercising the intended software against virtualized HW components. This enables to perform estimations on the temporal and spatial requirements of the different tasks of the system. In the case of the PS-TTM, the benefit of using virtual platform simulators would be double.

On one hand, being able to estimate the amount of time required by a job to be executed on a specific HW component would enable to reduce the uncertainty of its logical execution time. On the other hand, using an open-source platform simulator like the Open Virtual Platforms (OVP) [wwws], which provides a wide library of models for different processors and peripherals, would give the test engineers the chance to perform fault injection at low abstraction levels.

- *Graphical modeling*: Typing a whole time-triggered system design in plain text might become a tedious work. Being able to build the structure of the system by means of a graphical modeling tool would relax the complexity of the design. The graphical modeling tool presented in chapter 6, which addresses this issue, is still under development.

- *Reusability of test cases*: The reusability of test cases and fault injection campaigns over diverse abstraction levels until the hardware-in-the-loop testing phase for the final prototype dramatically reduces the effort needed for verification and validation. Although this work enables reusing test cases and FI campaigns in simulation (PIMs and PSMs), the transition from simulation-level to HiL testing often requires a refinement of the tests. Besides, other problems may arise, such as clock synchronization between different devices. The PhD research work by Carlos F. Nicolas (currently under development) aims to overcome these difficulties.

- *Parallel simulation*: The simulation of modern embedded systems demands a big amount of computational resources due to their complexity. The case-study of this thesis has shown the capability of the PS-TTM engine to simulate time-triggered systems with manageable simulation times. However, more detailed models would have required longer simulation times, thus increasing validation costs. Current SystemC simulators are based on the exploitation of a single-core machine, which makes them unable to take advantage of the parallelization capabilities of the nowadays widely extended multi/many-core platforms. Thus, the development of a parallel-SystemC engine would allow designers to reduce development time and costs by accelerating simulation. In that case, a multi-core version of PS-TTM running over the parallel SystemC engine would also take advantage of the capabilities of multi-core platforms, improving its simulation performance and reducing validation costs.

CHAPTER 9

List of Acronyms

AADL	Architecture, Analysis and Design Language
AMS	Analog Mixed Signal
ATE	Automatic Test Executor
BTM	Balise Transmission Module
CCS	Concurrent and Comparative Simulation
CNI	Communication Network Interface
CORBA	Common Object Request Broker Architecture
CP	Configuration and Planning
CPU	Central Processing Unit
CPS	Cyber-Physical System
CT	Continuous Time
DAL	Distributed Application Layer
DAS	Distributed Application Subsystem
DE	Discrete Event
DM	Diagnostic and Maintenance
DMI	Driver Machine Interface

DT Discrete Time

DOL Distributed Operation Layer

ECU Electronic Control Unit

ERTMS European Railway Traffic Management System

ESL Electronic System Level

ETCS European Train Control System

E-TTM Executable Time-Triggered Model

EVC European Vital Computer

FC Fault Configuration

FCR Fault Containment Region

FI Fault Injection

FIU Fault Injection Unit

FMEA Failure Mode and Effect Analysis

ForSyDe Formal System Design

FSM Finite State Machine

FTA Fault Tree Analysis

FTM Fault-Tolerance Mechanism

FTOS Fault Tolerant Operating System

GSM/R Global System for Mobile Communications - Railway

HetMoC Heterogeneous Model of Computation

HetSC Heterogeneous Specifications in SystemC

HTL Hierarchical Timing Language

HW Hardware

HWFI Hardware Based Fault Injection

IC Integrated Circuit

IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
JRU	Juridical Recorder Unit
KPN	Kahn Process Network
LET	Logical Execution Time
LTM	Loop Transmission Module
MARTE	Modeling and Analysis of Real-time Embedded Systems
MBD	Model Based Design
MBT	Model Based Testing
MDA	Model Driven Architecture
MDD	Model Driven Development
MDE	Model Driven Engineering
MDVP	Multi Domain Virtual Prototypes
ML	Meta-language
MoC	Model of Computation
MOGENTES	Model-Based Generation of Test-Cases for Embedded Systems
OMG	Object Management Group
PDM	Platform Definition Model
PIM	Platform Independent Model
PI-TTM	Platform Independent Time-Triggered Model
PM	Platform Model
PN	Process Network
PSM	Platform Specific Model
PS-TTM	Platform Specific Time-Triggered Model
RT	Real-Time

RTL	Register Transfer Level
SCADE	Safety-Critical Application Development Environment
SDF	Synchronous Dataflow
SEU	Single Event Upset
SFI	Simulated Fault Injection
SIL	Safety Integrity Level
SLDL	System-Level Design Language
SML	Standard ML
SML-Sys	Standard ML-Systems
SR	Synchronous Reactive
SUT	System Under Test
SW	Software
SWFI	Software Based Fault Injection
SysML	Systems Modeling Language
TCI	Test Case Interpreter
TDL	Timing Definition Language
TIU	Train Interface Unit
TLM	Transaction Level Model
TMO	Time-triggered Message-triggered Object
TMR	Triple Module Redundancy
TPM	Test Point Manager
TT	Time-Triggered
TTA	Time-Triggered Architecture
TTE	Time-Triggered Ethernet
TTP	Time-Triggered Protocol

UML Unified Modeling Language

VHDL VHSIC Hardware Description Language

VHSIC Very High Speed Integrated Circuit

WCET Worst-Case Execution Time

Bibliography

[AaB12] Mikel Azkarate-askasua Blazquez. *The Transient Tolerant Time-Triggered System-on-Chip (4TSoC)*. PhD thesis, Technische Universität Wien, 2012.

[Aer09] SAE Aerospace. SAE Architecture Analysis and Design Language (AADL), 2009.

[Aer11a] SAE Aerospace. Architecture Analysis and Design Language (AADL) Annex Volume 3: Annex E: Error Model Annex (draft 0.2), 2011.

[Aer11b] SAE Aerospace. Time-Triggered Ethernet, 2011-11-01 2011.

[Aer11c] SAE Aerospace. TTP Communication Protocol, 2011-02-08 2011.

[AH08] Per Andersson and Martin Höst. UML and SystemC – A Comparison and Mapping Rules for Automatic Code Generation. In Eugenio Villar, editor, *Embedded Systems Specification and Design Languages*, volume 10 of *Lecture Notes in Electrical Engineering*, pages 199–209. Springer Netherlands, 2008.

[ALR01] Algirdas Avizienis, Jean-Claude Laprie, and Brian Randell. Fundamental Concepts of Dependability. Technical report, University of Newcastle upon Tyne, 2001.

[ALRL04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, 2004.

[AMS07] Charles André, Frédéric Mallet, and Robert De Simone. Time Modeling in MARTE. In *Forum on specification & Design Languages (FDL)*, 2007.

[ANPP14] Iban Ayestaran, Carlos Fernando Nicolas, Jon Perez, and Peter Puschner. Modeling Logical Execution Time Based Safety-Critical Embedded Systems in SystemC. In *Embedded Computing (MECO), 2014 3rd Mediterranean Conference on*, pages 77–80, 2014.

[BB91] Albert Benveniste and Gérard Berry. The Synchronous Approach to Reactive and Real-Time Systems. In *Proceedings of the IEEE*, volume 79, pages 1270–1282. IEEE, 1991.

[BBC+09] Bernard Berthomieu, Jean-Paul Bodeveix, Christelle Chaudet, Silvano Zilio, Mamoun Filali, and Francois Vernadat. Formal Verification of AADL Specifications in the Topcased Environment. In *Proceedings of the 14th Ada-Europe International Conference on Reliable Software Technologies*, pages 207–221, 1573558, 2009. Springer-Verlag.

[BBD+14] Loïc Besnard, Etienne Borde, Pierre Dissaux, Thierry Gautier, Paul Le Guernic, and Jean-Pierre Talpin. Logically Timed Specifications in the AADL : a Synchronous Model of Computation and Communication (recommendations to the SAE committee on AADL). Technical report, Inria, 2014-04-02 2014.

[BBF+08] Giovanni Beltrame, Cristiana Bolchini, Luca Fossati, Antonio Miele, and Donatella Sciuto. ReSP: A non-intrusive Transaction-Level Reflective MPSoC Simulation Platform for Design Space Exploration. In *Design Automation Conference, 2008. ASPDAC 2008. Asia and South Pacific*, pages 673–678, 2008.

[BBH+07] Iuliana Bacivarov, Michael Beckinger, Wolfgang Haid, Kai Huang, and Lothar Thiele. Distributed Operation Layer: Optimal Mapping of Parallel Applications onto Heterogeneous Multiprocessor Tile-Based Architectures, Apr 2007.

[BCE+03] Albert Benveniste, Paul Caspi, Stephen Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The Synchronous Languages 12 Years Later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[BCG+97] Felice Balarin, Massimiliano Chiodo, Paolo Giusto, Harry Hsieh, Attila Jurecska, Luciano Lavagno, Claudio Passerone, Alberto Sangiovanni-Vincentelli, Ellen Sentovich, Kei Suzuki, and Bassam Tabbara. *Hardware-Software Co-Design of Embedded Systems: the POLIS Approach.* Kluwer Academic Publishers, 1997.

[BCR+11] M. Broy, S. Chakraborty, S. Ramesh, M. Satpathy, S. Resmerita, and W. Pree. Cross-layer Analysis, Testing and Verification of Automotive

Control Software. In *Embedded Software (EMSOFT), 2011 Proceedings of the International Conference on*, pages 263–272, 2011.

[Ber98] Gérard Berry. The Foundations of Esterel. In *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, page 31, 1998.

[Ber00] Gérard Berry. *The Esterel v5 Language Primer - Version v5.91*. Inria, 2000.

[BG92] Gérard Berry and Georges Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19:51, 1992.

[BGP07] Christian Bunse, Hans-Gerhard Gross, and Christian Peper. Applying a Model-based Approach for Embedded System Development. In *EUROMICRO 2007: Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications*, 2007.

[BKSZN10] Christian Buckl, Alois Knoll, Ina Schieferdecker, and Justyna Zander-Nowicka. Model-Based Analysis and Development of Dependable Systems. In Holger Giese, Gabor Karsai, Bernhard Rumpe, and Bernhard Schätz, editors, *Model-Based Engineering of Embedded Real-Time Systems*, Lecture Notes in Computer Science (LNCS), pages 271–293, Berlin Heidelberg New York, 2010. Springer.

[BLL$^+$07a] Christopher Brooks, Edward A. Lee, Xiaojun Liu, Stephen Neuendorffer, Yang Zhao, and Haiyang Zheng. Heterogeneous Concurrent Modeling and Design in Java (Volume 1: Introduction to Ptolemy II). Technical Report UCB/EECS-2007-7, EECS Department, University of California, Berkeley, January 11 2007.

[BLL$^+$07b] Christopher Brooks, Edward A. Lee, Xiaojun Liu, Stephen Neuendorffer, Yang Zhao, and Haiyang Zheng. Heterogeneous Concurrent Modeling and Design in Java (Volume 2: Ptolemy II Software Architecture). Technical Report UCB/EECS-2007-8, EECS Department, University of California, Berkeley, January 11 2007.

[BLL$^+$08] Christopher Brooks, Edward A. Lee, Xiaojun Liu, Stephen Neuendorffer, Yang Zhao, and Haiyang Zheng. Heterogeneous Concurrent Modeling and Design in Java (Volume 3: Ptolemy II Domains). Technical Report UCB/EECS-2008-37, EECS Department, University of California, Berkeley, April 15 2008.

[BMS08] Cristiana Bolchini, Antonio Miele, and Donatella Sciuto. Fault Models and Injection Strategies in SystemC Specifications. In *Digital System Design Architectures, Methods and Tools, 2008. DSD '08. 11th EUROMICRO Conference on*, pages 88–95, 2008.

[BP03] Alfredo Benso and Paolo Prinetto. *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*. Kluwer Academic Publishers, 2003.

[BPC98] J. Boue, P. Petillon, and Y. Crouzet. MEFISTO-L: A VHDL-Based Fault Injection Tool for the Experimental Assessment of Fault Tolerance. In *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*, pages 168–173, 1998.

[Bro06] Manfred Broy. Challenges in Automotive Software Engineering. In *Proceedings of the 28th international conference on Software engineering*, pages 33–42, 1134292, 2006. ACM.

[BSI98] British Standards Institution BSI. Software Testing. Software Component Testing , 15 August 1998 1998.

[Buc08] Christian Buckl. *Model-Based Development of Fault-Tolerant Real-Time Systems*. Phd thesis, Technische Universität München, 2008.

[But03] J.C. Butcher. *Numerical Methods for Ordinary Differential Equations*. Wiley, 2003.

[BWH$^+$03] Felice Balarin, Yosinori Watanabe, Harry Hsieh, Luciano Lavagno, Claudio Passerone, and Alberto Sangiovanni-Vincentelli. Metropolis: An Integrated Electronic System Design Environment. *Computer*, 36(4):45–52, 2003.

[CEN11] CENELEC. EN 50128: Railway Applications — Software for Railway Control and Protection Systems, 2011.

[Cha85] Daniel Marcos Chapiro. *Globally-Asynchronous Locally-Synchronous Systems (Performance, Reliability, Digital)*. PhD thesis, Stanford University, 1985.

[Chr14] Chrona. TDL Specification and Language Report. Technical report, Chrona, 2014. Version 1.6.

[CK06] F.E. Cellier and E. Kofman. *Continuous System Simulation*. Springer, 2006.

[Cow01] Nelson Cowan. The Magical Number 4 in Short-Term Memory: A Reconsideration of Mental Storage Capacity. *Behavioral and Brain Sciences*, 24(01):87–114, 2001.

[CRM^{+}09] Alfons Crespo, Ismael Ripoll, Miguel Masmano, P. Arberet, and J.J. Metge. XtratuM: An Open Source Hypervisor for TSP Embedded Systems in Aerospace. In *DASIA 2009*, 2009.

[DES01] DESS. The DESS Methodology. Technical report, ITEA, December 2001. Version 01.

[Dij70] Edsger W. Dijkstra. Notes on Structured Programming. circulated privately, 1970.

[DNP^{+}10] Patricia Derler, Andreas Naderlinger, Wolfgang Pree, Stefan Resmerita, and Josef Templ. Simulation of LET Models in Simulink and Ptolemy. In Christine Choppy and Oleg Sokolsky, editors, *Foundations of Computer Software. Future Trends and Techniques for Development*, volume 6028 of *Lecture Notes in Computer Science*, pages 83–92. Springer Berlin Heidelberg, 2010.

[DNSV10] M. Di Natale and Alberto Sangiovanni-Vincentelli. Moving From Federated to Integrated Architectures in Automotive: The Role of Standards, Methods and Tools. *Proceedings of the IEEE*, 98(4):603–620, 2010.

[EBF05] Jacky Estublier, Jean Bezivin, and Jean-Marie Favre. Model Driven Engineering: Myths, Reality and Potential, 2005.

[EBK03] Wilfried Elmenreich, Günther Bauer, and Hermann Kopetz. The Time-Triggered Paradigm. *Yearbook of Morphology*, 2003.

[EJ09] C. Ebert and C. Jones. Embedded Software: Facts, Figures, and Future. *Computer*, 42(4):42–52, 2009.

[EJL^{+}03] Johan Eker, Jörn W. Janneck, Edward A. Lee, Liu Jie, Liu Xiaojun, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Xiong Yuhong. Taming Heterogeneity - The Ptolemy Approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[ELLSV97] Stephen Edwards, Luciano Lavagno, Edward A. Lee, and Alberto Sangiovanni-Vincentelli. Design of Embedded Systems: Formal Models, Validation, and Synthesis. *Proceedings of the IEEE*, 85(3):366–390, 1997.

[Enc03] Vincent Encontre. Testing Embedded Systems: Do You Have the GuTs for it? `http://www.ibm.com/developerworks/rational/library/459.html`, 2003. Accessed: 02/05/2014.

[Fis01] George S. Fishman. *Discrete-Event Simulation: Modeling, Programming, and Analysis.* Springer, 2001.

[FSK98] P. Folkesson, S. Svensson, and J. Karlsson. A Comparison of Simulation Based and Scan Chain Implemented Fault Injection. In *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*, pages 284–293, 1998.

[FWI$^+$14] Victor Fernández, Elier Wilpert, Henrique Isidoro, Cédric Ben Aoun, and François Pêcheux. SystemC-MDVP Modelling of Pressure Driven Microfluidic Systems. In *3rd Mediterranean Conference on Embedded Computing (MECO)*, 2014.

[GAGS09] Daniel D. Gajski, Samar Abdi, Andreas Gerstlauer, and Gunar Schirner. *Embedded System Design: Modeling, Synthesis and Verification.* Springer Publishing Company, Incorporated, 2009.

[GBGG01] Joaquin Gracia, Juan Carlos Baraza, Daniel Gil, and Pedro Jose Gil. Comparison and Application of different VHDL-Based Fault Injection Techniques. In *Defect and Fault Tolerance in VLSI Systems, 2001. Proceedings. 2001 IEEE International Symposium on*, pages 233–241, 2001.

[GETS10] Sébastien Gérard, Huascar Espinoza, François Terrier, and Bran Selic. Modeling Languages for Real-Time and Embedded Systems. In Holger Giese, Gabor Karsai, Edward Lee, Bernhard Rumpe, and Bernhard Schätz, editors, *Model Based Engineering of Embedded Real-Time Systems*, pages 129–154. Springer, 2010.

[Ghe06] Frank Ghenassia. *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems.* Springer, 2006.

[GK83] Daniel D. Gajski and Robert H. Kuhn. Guest Editors' Introduction: New VLSI Tools. *Computer*, 16(12):11–14, 1983.

[GKL$^+$10] Holger Giese, Gabor Karsai, Edward A. Lee, Bernhard Rumpe, and Bernhard Schätz. *Model-Based Engineering of Embedded Real-Time Systems*, volume LNCS 6100 of *Lecture Notes in Computer Science (LNCS).* Springer, Berlin Heidelberg NewYork, 2010.

[GSVK$^+$06] Arkadeb Ghosal, Alberto Sangiovanni-Vincentelli, Christoph M. Kirsch, Thomas A. Henzinger, and Daniel Iercan. A Hierarchical Coordination Language for Interacting Real-Time Tasks. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 132–141, 1176907, 2006. ACM.

[Hal05] Nicolas Halbwachs. A Synchronous Language at Work: the Story of Lustre. In *Formal Methods and Models for Co-Design, 2005. MEMOCODE '05. Proceedings. Third ACM and IEEE International Conference on*, pages 3–11, 2005.

[Ham14] Felicia Hamerman. 2014 Embedded Market Study. Technical report, UBM Tech, 2014.

[HCRP91] Nicolas Halbwachs, Paul Caspi, P Raymond, and D. Pilaud. The Synchronous Data Flow Programming Language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[Hen04] Thomas A. Henzinger. Embedded Software: Better Models, Better Code. In Jordi Cortadella and Wolfgang Reisig, editors, *Applications and Theory of Petri Nets 2004*, volume 3099 of *Lecture Notes in Computer Science*, pages 35–36. Springer Berlin Heidelberg, 2004.

[HHBT09] Wolfgang Haid, Kai Huang, Iuliana Bacivarov, and Lothar Thiele. Multiprocessor SoC Software Design Flows. *IEEE Signal Processing Magazine*, 26(6):64–71, 2009.

[HHK01a] Thomas A. Henzinger, Benjamin Horowitz, and Christoph M. Kirsch. Embedded Control Systems Development with Giotto. In *Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems*, pages 64–72, 384208, 2001. ACM.

[HHK01b] Thomas A. Henzinger, Benjamin Horowitz, and Christoph M. Kirsch. Giotto: A Time-Triggered Language for Embedded Programming. In Thomas Henzinger and Christoph Kirsch, editors, *Embedded Software*, volume 2211 of *Lecture Notes in Computer Science*, pages 166–184. Springer Berlin / Heidelberg, 2001.

[HHK03] Thomas Henzinger, Benjamin Horowitz, and Christoph Kirsch. Giotto: A Time-Triggered Language for Embedded Programming. *Proceedings of the IEEE*, 91(1):84–99, 2003.

[HK02] Thomas A. Henzinger and Christoph M. Kirsch. The Embedded Machine: Predictable, Portable Real-Time Code . *ACM SIGPLAN Conf. Program. Lang. Design Implementation*, pages 315–326, 2002.

[HKMS09] Thomas A. Henzinger, Christoph M. Kirsch, Eduardo R.B. Marques, and Ana Sokolova. Distributed, Modular HTL. In *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pages 171–180, 2009.

[HKSP03] Thomas A. Henzinger, Christoph M. Kirsch, Marco Sanvido, and Wolfgang Pree. From Control Models to Real-Time Code Using Giotto. *Control Systems, IEEE*, 23(1):50–64, 2003.

[HLL$^+$03] Christopher Hylands, Edward A. Lee, Jie Liu, Xiaojun Liu, Stephen Neuendorffer, Yuhong Xiong, Yang Zhao, and Haiyang Zheng. Overview of the Ptolemy Project. Technical report, UC Berkeley, 2/07/2003 2003.

[HM06] Nicolas Halbwachs and L. Mandel. Simulation and Verification of Asynchronous Systems by means of a Synchronous Model. In *Application of Concurrency to System Design, 2006. ACSD 2006. Sixth International Conference on*, pages 3–14, 2006.

[HNSS09] Hans Hansson, Thomas Nolte, Mikael Sjödin, and Daniel Sundmark. Real-Time in Networked Embedded Systems. In *Embedded Systems Design and Verification*. CRC Press, 2009.

[Hoa96] C. A. R. Hoare. How Did Software Get So Reliable Without Proof? In Marie-Claude Gaudel and James Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *Lecture Notes in Computer Science*, pages 1–17. Springer Berlin Heidelberg, 1996.

[Hoe05] Carl Hoefer. Causality and Determinism: Tension, or Outright Conflict? *Revista de Filosofía*, 29(2), 2005.

[HS07] Thomas A. Henzinger and Joseph Sifakis. The Discipline of Embedded Systems Design. *Computer*, 40(10):32–40, 2007.

[HSV05] Fernando Herrera, Pablo Sánchez, and Eugenio Villar. Heterogeneous System-Level Specification in SystemC. In Pierre Boulet, editor, *Advances in Design and Specification Languages for SoCs*, pages 199–216. Springer US, 2005.

[HTI97] Mei-Chen Hsueh, Timothy Tsai, and Ravishankar Iyer. Fault Injection Techniques and Tools. *IEEE*, page 8, 1997.

[HV06] Fernando Herrera and Eugenio Villar. A Framework for Embedded System Specification under Different Models of Computation in SystemC. In *Design Automation Conference, 2006 43rd ACM/IEEE*, pages 911–914, 2006.

[HV08] Fernando Herrera and Eugenio Villar. A Framework for Heterogeneous Specification and Design of Electronic Embedded Systems in SystemC. *ACM Trans. Des. Autom. Electron. Syst.*, 12(3):1–31, 2008.

[IEC10] IEC 61508: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems, 2010.

[IEE91] IEEE Standard Computer Dictionary. A Compilation of IEEE Standard Computer Glossaries, 1991.

[IEE05] IEEE Standard SystemC® Language Reference Manual, 2005.

[IEE06] IEEE Standard for Verilog Hardware Description Language, 2006.

[IEE09] IEEE Standard VHDL Language Reference Manual, 2009.

[IEE13] Standard SystemC® AMS extensions 2.0 Language Reference Manual, 2013.

[ISO09] ISO/DIS 26262: Road Vehicles — Functional Safety, 2009.

[IST14] International Software Testing Qualifications Board ISTQB. Standard Glossary of Terms Used in Software Testing. Technical report, ISTQB, March 28th, 2014 2014. Version 2.3.

[Jan09] Axel Jantsch. Models of Computation for Distributed Embedded Systems. In *Embedded Systems Design and Verification*. CRC Press, 2009.

[JAR$^+$94] Eric Jenn, Jean Arlat, Marcus Rimen, Joakim Ohlsson, and Johan Karlsson. Fault Injection into VHDL Models: The MEFISTO Tool. In *Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on*, pages 66–75, 1994.

[JDR09] M. Jeitler, M. Delvai, and S. Reichor. FuSE - a Hardware Accelerated HDL Fault Injection Tool. In *Programmable Logic, 2009. SPL. 5th Southern Conference on*, pages 89–94, 2009.

[Joz01] Lech Jozwiak. Quality-Driven Design in the System-On-a-Chip Era: Why and How? *Journal of Systems Architecture*, 47(3–4):201–224, 2001.

[JS05] A. Jantsch and I. Sander. Models of Computation and Languages for Embedded System Design. *Computers and Digital Techniques, IEE Proceedings -*, 152(2):114–129, 2005.

[JSEB04] I. Johnson, C. Snook, A. Edmunds, and M. Butler. Rigorous Development of Reusable, Domain-Specific Components, for Complex Applications. In *In CSDUML'04 - 3rd International Workshop on Critical Systems Development with UML*, 2004.

[JSPP04] Mirko Jakovljevic, Martin Schlager, Markus Plankensteiner, and Stefan Poledna. Safety Relevant Automotive Electronic Solutions. *Automotive Electronics International*, March:21–23, 2004.

[JTM07] Daniel Jackson, Martyn Thomas, and Lynette I. Millett. *Software for Dependable Systems: Sufficient Evidence?* The National Academies Press, 2007.

[KAGS05] Hermann Kopetz, Astrit Ademaj, Petr Grillinger, and Kklaus Steinhammer. The Time-Triggered Ethernet (TTE) Design. In *Object-Oriented Real-Time Distributed Computing, 2005. ISORC 2005. Eighth IEEE International Symposium on*, pages 22–33, 2005.

[Kah74] Gilles Kahn. The Semantics of Simple Language for Parallel Programming. In *IFIP Congress*, pages 471–475, conf/ifip/Kahn74, 1974.

[KAVS08] Ali Koudri, Denis Aulagnier, Didier Vojtisek, and Philippe Soulard. Using MARTE in a Co-Design Methodology. In *Design, Automation and Test in Europe (DATE)*, 2008.

[KB03] Hermann Kopetz and Günther Bauer. The Time-Triggered Architecture. In *Proceedings of the IEEE*, volume 91, page 15, 2003.

[KDvdWV02] Bart Kienhuis, Ed Deprettere, Pieter van der Wolf, and Kees Vissers. A Methodology to Design Programmable Embedded Systems - The Y-Chart Approach. In *Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation - SAMOS*, pages 18–37, 691571, 2002. Springer-Verlag.

[KDVvdW97] Bart Kienhuis, Ed Deprettere, Kees Vissers, and Pieter van der Wolf. An Approach for Quantitative Analysis of Application-Specific Dataflow Architectures . In *Application-Specific Systems, Architectures and Processors, 1997. Proceedings., IEEE International Conference on*, pages 338–349, 1997.

[KFFM10] Ingolf Krüger, Claudiu Farcas, Emilia Farcas, and Massimiliano Menarini. Requirements Modeling for Embedded Realtime Systems. In Holger Giese, Gabor Karsai, Edward Lee, Bernhard Rumpe, and Bernhard Schätz, editors, *Model Based Engineering of Embedded Real-Time Systems*, pages 155–199. Springer, 2010.

[KG93] Hermann Kopetz and Günter Grunsteidl. TTP - A Time-Triggered Protocol for Fault-Tolerant Real-Time Systems. In *Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers., The Twenty-Third International Symposium on*, pages 524–533, 1993.

[KGGV07] Milos Krstic, Eckhard Grass, Frank K. Gurkaynak, and Pascal Vivet. Globally Asynchronous, Locally Synchronous Circuits: Overview and Outlook. *Design & Test of Computers, IEEE*, 24(5):430–441, 2007.

[Kim97] K. H. Kim. Object Structures for Real-Time Systems and Simulators. *Computer*, 30(8):62–70, 1997.

[KLM08] Christoph Kirsch, Luis Lopes, and Eduardo R.B. Marques. Semantics-Preserving and Incremental Runtime Patching of Real-Time Programs. In *Proc. ARTIST Workshop on Adaptive and Reconfigurable Embedded Systems (APRES)*, 2008.

[KN97] Hermann Kopetz and Roman Nossal. Temporal Firewalls in Large Distributed Real-Time Systems. In *Distributed Computing Systems, 1997., Proceedings of the Sixth IEEE Computer Society Workshop on Future Trends of*, pages 310–315, 1997.

[KOESH07] Hermann Kopetz, Roman Obermaisser, Christian El Salloum, and Bernhard Huber. Automotive Software Development for a Multi-Core System-on-a-Chip. In *Software Engineering for Automotive Systems, 2007. ICSE Workshops SEAS '07. Fourth International Workshop on*, pages 2–2, 2007.

[Kop92] Hermann Kopetz. Sparse Time versus Dense Time in Distributed Real-Time Systems. In *Distributed Computing Systems, 1992., Proceedings of the 12th International Conference on*, pages 460–467, 1992.

[Kop98a] Hermann Kopetz. The Time-Triggered Architecture. In *Object-Oriented Real-time Distributed Computing, 1998. (ISORC 98)*, page 8, 1998.

[Kop98b] Hermann Kopetz. The Time-Triggered Model of Computation. *In Real Time Systems Symposium, IEEE Computer Society*, page 16, 1998.

[Kop06] Hermann Kopetz. On the Fault Hypothesis for a Safety-Critical Real-Time System. In *Automotive Software – Connected Services in Mobile Networks*, page 8. Springer, 2006.

[Kop08] Hermann Kopetz. The Complexity Challenge in Embedded System Design. In *ISORC*, page 10. IEEE, 2008.

[Kop11] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer, 2011.

[KS03] Hermann Kopetz and Neeraj Suri. Compositional Design of RT Systems: A Conceptual Basis for Specification of Linking Interfaces. In *Object-Oriented Real-Time Distributed Computing, 2003. Sixth IEEE International Symposium on*, pages 51–60, 2003.

[KS12] Christoph M. Kirsch and Ana Sokolova. The Logical Execution Time Paradigm. In *Advances in Real-Time Systems*, pages 103–120. Springer Berlin Heidelberg, 2012.

[Kur05] Ivan Kurtev. *Adaptability of Model Transformations*. Phd, University of Twente, 2005.

[Kyu08] Kita Kyushu. Clock Constraint Specification Language. Technical report, Inria, 2008.

[LAK92] Jean-Claude Laprie, Algirdas Avizienis, and Hermann Kopetz. *Dependability: Basic Concepts and Terminology*. Springer-Verlag New York, Inc., 1992.

[Lap85] Jean-Claude Laprie. Dependable Computing and Fault Tolerance: Concepts and Terminology. In *International Symposion on Fault Tolerant Computing Systems*, 1985.

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

[Lee02] Edward A. Lee. Embedded Software. In V. Zelkowitz Marvin, editor, *Advances in Computers*, volume Volume 56, pages 55–95. Elsevier, 2002.

[Lee08] Edward A. Lee. Cyber Physical Systems: Design Challenges. Technical Report UCB/EECS-2008-8, EECS Department, University of California, Berkeley, January 23 2008.

[LGGLBLM91] Paul Le Guernic, Thierry Gautier, Michel Le Borgne, and Claude Le Maire. Programming Real-Time Applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, 1991.

[LH94] J. H. Lala and R. E. Harper. Architectural Principles for Safety-Critical Real-Time Applications. *Proceedings of the IEEE*, 82(1):25–40, 1994.

[LN05] Edward A. Lee and Stephen Neuendorffer. Concurrent Models of Computation for Embedded Software. *Computers and Digital Techniques, IEE Proceedings* -, 152(2):239–250, 2005.

[LP02] Edward A. Lee and Thomas M. Parks. Dataflow Process Networks. In Micheli Giovanni De, Ernst Rolf, and Wolf Wayne, editors, *Readings in hardware/software co-design*, pages 59–85. Kluwer Academic Publishers, 2002.

[LR11] Weiyun Lu and Martin Radetzki. Efficient Fault Simulation of SystemC Designs. In *Digital System Design (DSD), 2011 14th Euromicro Conference on*, pages 487–494, 2011.

[LS11] Edward A. Lee and Sanjit A. Seshia. *Introduction to Embedded Systems: A Cyber-Physical Systems Approach.* UC Berkeley, Berkeley, 1 edition, 2011.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[LSV98] Edward A. Lee and Alberto Sangiovanni-Vincentelli. A Framework for Comparing Models of Computation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 17(12):1217–1229, 1998.

[LSVS98] Luciano Lavagno, Alberto Sangiovanni-Vincentelli, and Ellen Sentovich. Models of Computation for Embedded System Design. In Ahmed A. Jerraya and Jean Mermet, editors, *System-Level Synthesis*, pages 45–102. Kluwer Academic Publishers, 1998.

[LTS$^+$09] James Lapalme, Bart Theelen, Nikolay Stoimenov, Jeroen Voeten, Lothar Thiele, and El Mostapha Aboulhamid. Y-Chart Based System Design: A Discussion on Approaches. In *Nouvelles approches pour la conception d'outils CAO pour le domaine des systems embarqu'es*, 2009.

[MAR10] Monica Malvezzi, Benedetto Allotta, and Mirko Rinchi. Odometric Estimation for Automatic Train Protection and Control Systems. *Vehicle System Dynamics*, 49(5):723–739, 2010.

[Mar11] Peter Marwedel. *Embedded System Design*. Springer, 2nd edition, 2011.

[Mil83] Robin Milner. Calculi for Synchrony and Asynchrony. *Theoretical Computer Science*, 25(3):267–310, 1983.

[Mil97] Robin Milner. *The Definition of Standard ML: Revised*. MIT Press, 1997.

[MM03] Joaquin Miller and Jishnu Mukerji. MDA Guide Version 1.0.1, 2003/06/12 2003.

[MMB09] Robin E. McDermott, Raymond J. Mikulak, and Michael R. Beauregard. *The Basics of FMEA*. CRC Press, 2 edition, 2009.

[MMP09] L. G. Murillo, Marcello Mura, and M. Prevostini. Semi-Automated HW/SW Co-Design for Embedded Systems: From MARTE Models to SystemC Simulators. In *Forum on Specification & Design Languages (FDL)*, pages 1–6, 2009.

[MOG09] MOGENTES. Fault Models. Technical report, MOGENTES, 2009/12/29 2009.

[Moo65] Gordon E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8), 1965.

[MPS04] Deepak Mathaikutty, Hiren D. Patel, and Sandeep K. Shukla. A Functional Programming Framework of Heterogeneous Model of Computation for System Design. In *Forum on Design Languages (FDL)*, pages 586–598, 2004.

[MPSJ06] Deepak A. Mathaikutty, Hiren D. Patel, Sandeep K. Shukla, and Axel Jantsch. UMoC++: A C++-Based Multi-MoC Modeling Environment. In A. Vachoux, editor, *Applications of Specification and Design Languages for SoCs*, pages 115–130. Springer Netherlands, 2006.

[MPSJ08a] Deepak Mathaikutty, Hiren Patel, Sandeep Shukla, and Axel Jantsch. EWD: A Metamodeling Driven Customizable Multi-MoC System Modeling Framework. *ACM Trans. Des. Autom. Electron. Syst.*, 12(3):1–43, 2008.

[MPSJ08b] Deepak A. Mathaikutty, Hiren D. Patel, Sandeep K. Shukla, and Axel Jantsch. SML-Sys: a Functional Framework with Multiple Models of Computation for Modeling Heterogeneous System. *Design Automation for Embedded Systems*, 12(1-2):1–30, 2008.

[MRCP10] Miguel Masmano, Ismael Ripoll, Alfons Crespo, and Salvador Peiro. XtratuM for LEON3: an Open Source Hypervisor for High Integrity Systems. In *ERTS-2010*, 2010.

[MVS07] Silvio Misera, Heinrich Theodor Vierhaus, and André Sieber. Fault Injection Techniques and their Accelerated Simulation in SystemC. In *Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on*, pages 587–595, 2007.

[Nag75] Laurence W. Nagel. *SPICE2: A Computer Program to Simulate Semiconductor Circuits*. PhD thesis, EECS Department, University of California, Berkeley, 1975.

[OESHK07] Roman Obermaisser, Christian El Salloum, Bernhard Huber, and Hermann Kopetz. Modeling and Verification of Distributed Real-Time Systems using Periodic Finite State Machines. *Journal of Computer Systems Science & Engineering*, 2007.

[OMG11a] Object Management Group OMG. OMG Unified Modeling Language (OMG UML), 2011 - 08 - 05 2011. Version 2.4.1.

[OMG11b] Object Management Group OMG. UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, 2011-06-02 2011. Version 1.1.

[OMG12] Object Management Group OMG. OMG Systems Modeling Language (OMG SysML), 2012 - 06 - 01 2012. Version 1.3.

[Oxf07] Dictionaries Oxford. Oxford English Dictionary, 2007.

[PAaP10] Jon Perez, Mikel Azkarate-askasua, and Antonio Perez. Codesign and Simulated Fault Injection of Safety-Critical Embedded Systems Using SystemC. In *European Dependable Computing Conference*, page 9, 2010.

[Per11] Jon Perez. *Executable Time-Triggered Model (E-TTM) for the Development of Safety-Critical Embedded Systems*. PhD thesis, Technischen Universität Wien, 2011.

[PHV12] Pablo Peñil, Fernando Herrera, and Eugenio Villar. Formal Support for Untimed MARTE-SystemC Interoperability. In Tom J. Kaźmierski and Adam Morawiec, editors, *System Specification and Design Languages*, volume 106 of *Lecture Notes in Electrical Engineering*, pages 239–254. Springer New York, 2012.

[PMPV10] P Peñil, J Medina, H Posadas, and E Villar. Generating Heterogeneous Executable Specifications in SystemC from UML/MARTE Models. *Innovations in Systems and Software Engineering*, 6(1-2):65–71, 2010.

[PNOES10a] Jon Perez, Carlos Fernando Nicolas, Roman Obermaisser, and Christian El Salloum. Modeling Time-Triggered Architecture Based Real-Time Systems Using SystemC. In Tom J. Kaźmierski and Adam Morawiec, editors, *System Specification and Design Languages: Selected Contributions from FDL 2010*, pages 123–142. Springer, 2010.

[PNOES10b] Jon Perez, Carlos Fernando Nicolas, Roman Obermaisser, and Christian El Salloum. Modeling Time-Triggered Architecture Based Safety-Critical Embedded Systems Using SystemC. In *Forum on specification & Design Languages (FDL)*, pages 1–6, 2010.

[PONA11] Jon Perez, Roman Obermaisser, Carlos Fernando Nicolas, and Iban Ayestaran. Modeling Time-Triggered Real-Time Control Systems Using Executable Time-Triggered Model (E-TTM) and SystemC-AMS. *Comput. Syst. Sci. Eng.*, 26(6), 2011.

[PPO10] Jon Perez, Antonio Perez, and Roman Obermaisser. Executable Time-Triggered Model (E-TTM) for Real-Time Control Systems. In *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2010 13th IEEE International Symposium on*, pages 42–49, 2010.

[PS04] Hiren D. Patel and Sandeep K. Shukla. Towards A Heterogeneous Simulation Kernel for System Level Models: A SystemC Kernel for Synchronous Data Flow Models. In *Proceedings of the 14th ACM Great Lakes symposium on VLSI*, pages 248–253. ACM, 2004.

[PS06] Hiren D. Patel and Sandeep K. Shukla. *SystemC Kernel Extensions for Heterogeneous System Modeling: A Framework for Multi-MoC Modeling & Simulation.* Kluwer Academic Publishers, 2006.

[Pto14] Claudius Ptolemaeus. *System Design, Modeling and Simulation using Ptolemy II.* Ptolemy.org, 1.02 edition, 2014.

[RDPN10] Stefan Resmerita, Patricia Derler, Wolfgang Pree, and Andreas Naderlinger. Modeling and Simulation of TDL Applications. In Holger Giese, Gabor Karsai, Edward Lee, Bernhard Rumpe, and Bernhard Schätz, editors, *Model-Based Engineering of Embedded Real-Time Systems*, volume 6100 of *Lecture Notes in Computer Science*, pages 107–128. Springer Berlin Heidelberg, 2010.

[RPV$^+$13] Sebastian Reiter, Michael Pressler, Alexander Viehl, Oliver Bringmann, and Wolfgang Rosenstiel. Reliability Assessment of Safety-Relevant Automotive Systems in a Model-Based Design Flow. In *Design Automation Conference (ASP-DAC), 2013 18th Asia and South Pacific*, pages 417–422, 2013.

[Rum06] B. Rumpler. Complexity Management for Composable Real-Time Systems. In *Object and Component-Oriented Real-Time Distributed Computing, 2006. ISORC 2006. Ninth IEEE International Symposium on*, page 9 pp., 2006.

[SBF$^+$07] Thomas Sporer, Michael Beckinger, Andreas Franck, Iuliana Bacivarov, Wolfgang Haid, Kai Huang, Lothar Thiele, Pier S. Paolucci, Piergiovanni Bazzana, Piero Vicini, Jianjiang Ceng, Stefan Kraemer, and Rainer Leupers. SHAPES - A Scalable Parallel HW/SW Architecture Applied to Wave Field Synthesis. In *Proc. 32nd Int'l Audio Engineering Society (AES) Conference*, pages 175–187, 2007.

[SBR$^+$12] Lars Schor, Iuliana Bacivarov, Devendra Rai, Hoeseok Yang, Shin-Haeng Kang, and Lothar Thiele. Scenario-Based Design Flow for Mapping Streaming Applications onto On-Chip Many-Core Systems. In *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems*, pages 71–80, 2380422, 2012. ACM.

[Sel03] Bran Selic. The Pragmatics of Model-Driven Development. *Software, IEEE*, 20(5):19–25, 2003.

[SJ04] Ingo Sander and Axel Jantsch. System Modeling and Transformational Design Refinement in ForSyDe. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 23(1):17–32, 2004.

[SK97] Janos Sztipanovits and Gabor Karsai. Model-Integrated Computing. *Computer*, 30(4):110–111, 1997.

[SK01] Janos Sztipanovits and Gabor Karsai. Embedded Software: Challenges and Opportunities. In ThomasA Henzinger and ChristophM Kirsch, editors, *Embedded Software*, volume 2211 of *Lecture Notes in Computer Science*, pages 403–415. Springer Berlin Heidelberg, 2001.

[SRAH08] Rishad Ahmed Shafik, Paul Rosinger, and Bashir Al-Hashimi. SystemC-based Minimum Intrusive Fault Injection Technique with Improved Fault Representation. *International On-line Test Symposium (IOLTS)*, page 6, 2008.

[STB96] Volkmar Sieh, Oliver Tschache, and Frank Balbach. VHDL-based Fault Injection with VERIFY. Technical report, Erlangen: Universität Erlangen-Nürnberg, 1996.

[STB97] Volkmar Sieh, Oliver Tschache, and Frank Balbach. VERIFY Evaluation of Reliability Using VHDL-Models with Embedded Fault Descriptions. In *Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers., Twenty-Seventh Annual International Symposium on*, pages 32–36, 1997.

[SV07] Alberto Sangiovanni-Vincentelli. Quo Vadis, SLD? Reasoning About the Trends and Challenges of System Level Design. *Proceedings of the IEEE*, 95(3):467–506, 2007.

[TLMP93] Shin-Yuan Tzou, Jyh-Jang Lim, Jai Menon, and David Palmer. A Distributed Development Environment for Embedded Software. *Software—Practice & Experience*, 23, 1993.

[TM02] D.E. Thomas and P.R. Moorby. *The Verilog® Hardware Description Language*. Springer, 2002.

[UL07] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., 2007.

[Van10] Steven H. VanderLeest. ARINC 653 Hypervisor. In *Digital Avionics Systems Conference (DASC), 2010 IEEE/AIAA 29th*, pages 5.E.2–1–5.E.2–20, 2010.

[WGR09] Peter Winter, Bettina Guiot, and International Union of Railways. *Compendium on ERTMS: European Rail Traffic Management System*. Eurail Press, 2009.

[wwwa] Accelera Systems Initiative. `http://www.accellera.org/home/`. Accessed: 2014/10/07.

[wwwb] Catrene (CA701) H-Inception. https://www-soc.lip6.fr/trac/hinception. Accessed: 2014-07-24.

[wwwc] Chrona Creation Suite. http://www.chrona.com/en/products/chrona-creation-suite/. Accessed: 2014/07/04.

[wwwd] CoFluent Studio. https://www.cofluentdesign.com/index.php/Products_Services/cofluent-studio.html. Accessed: 2014/06/27.

[wwwe] DALipse. http://www.dal.ethz.ch/index.php?title=DALipse. Accessed: 2015/01/02.

[wwwf] Distributed Application Layer (DAL). http://www.dal.ethz.ch/index.php?title=Distributed_Application_Layer_%28DAL%29. Accessed: 2015/01/02.

[wwwg] Distributed Operation Layer. http://www.tik.ee.ethz.ch/~shapes/dol.html. Accessed: 2014/06/26.

[wwwh] Dymola. http://www.3ds.com/products-services/catia/capabilities/systems-engineering/modelica-systems-simulation/dymola/. Accessed: 2014/06/30.

[wwwi] European Reference Tiled Architecture Experiment (EURETILE) project. www.euretile.eu. Accessed: 2015/01/02.

[wwwj] EZwave. http://www.mentor.com/products/ic_nanometer_design/analog-mixed-signal-verification/ezwave/. Accessed: 2014/10/23.

[wwwk] GTKWave. http://gtkwave.sourceforge.net/. Accessed: 2014/10/23.

[wwwl] HetSC. http://www.teisa.unican.es/HetSC/. Accessed: 2014/07/14.

[wwwm] Integrity Multivisor. http://www.ghs.com/products/rtos/integrity_virtualization.html. Accessed: 30-08-2014.

[wwwn] LibreOffice. https://www.libreoffice.org/. Accessed: 2015/01/03.

[wwwo] Matlab. https://es.mathworks.com/products/matlab/. Accessed: 2015/01/03.

[wwwp] Microsoft Office. https://office.microsoft.com/. Accessed: 2015/01/03.

[wwwq] Moses. http://www.tik.ee.ethz.ch/~shapes/moses.html. Accessed: 2011/12/22.

[wwwr] NI LabVIEW C Generator. http://sine.ni.com/nips/cds/view/p/lang/en/nid/209015. Accessed: 2014/06/26.

[wwws] Open Virtual Platforms. http://www.ovpworld.org/. Accessed: 2014/12/22.

[wwwt] Ptolemy II. http://ptolemy.eecs.berkeley.edu/ptolemyII/. Accessed: 201/06/26.

[wwwu] Python Programming Language. http://www.python.org/. Accessed: 2014/12/31.

[wwwv] SCADE Success Stories. http://www.esterel-technologies.com/success-stories/. Accessed: 2014/06/27.

[wwww] SCADE Suite. http://www.esterel-technologies.com/products/scade-suite/. Accessed: 2014-06-26.

[wwwx] Simulink. http://www.mathworks.es/products/simulink/. Accessed: 2014/06/30.

[wwwy] Simulink Coder. http://www.mathworks.es/products/simulink-coder/. Accessed: 2014/06/26.

[wwwz] SimVision. http://www.cadence.com/products/fv/simvision. Accessed: 201/10/23.

[ZAV04] Haissam Ziade, Rafic Ayoubi, and Raoul Velazco. A Survey on Fault Injection Techniques. *The International Arab Journal of Information Technology*, 1:16, 2004.

[ZME03] H. R. Zarandi, S. G. Miremadi, and A. Ejlali. Dependability Analysis Using a Fault Injection Tool Based on Synthesizability of HDL Models. In *Defect and Fault Tolerance in VLSI Systems, 2003. Proceedings. 18th IEEE International Symposium on*, pages 485–492, 2003.

[ZN08] Justyna Zander-Nowicka. *Model-Based Testing of Real-Time Embedded Systems in the Automotive Domain*. Phd thesis, Technische Universität Berlin, 2008.

[ZSJ10] Jun Zhu, Ingo Sander, and Axel Jantsch. HetMoC: Heterogeneous Modelling in SystemC. In *Specification & Design Languages (FDL 2010), 2010 Forum on*, pages 1–6, 2010.

Selected Publications

- Iban Ayestaran, Carlos F. Nicolas, Jon Perez, Asier Larrucea, Peter Puschner: **A Novel Modeling Framework for Time-Triggered Safety-Critical Embedded Systems**. *Forum on specification & Design Languages (FDL)*, 2014, Munich, Germany.

- Iban Ayestaran, Irune Agirre, Carlos F. Nicolas, Jon Perez: **Simulated Fault Injection for the Validation of Fault Tolerance Mechanisms in Dependable Time-Triggered Systems**. *V Jornadas de Computación Empotrada (JCE)*, 2014, Valladolid, Spain.

- Iban Ayestaran, Carlos F. Nicolas, Jon Perez, Asier Larrucea, Peter Puschner: **A Simulated Fault Injection Framework for Time-Triggered Safety-Critical Embedded Systems**. *The 33rd International Conference on Computer Safety, Reliability and Security (SafeComp)*, 2014, Firenze, Italy.

- Iban Ayestaran, Carlos F. Nicolas, Jon Perez, Peter Puschner: **Modeling Logical Execution Time Based Safety-Critical Embedded Systems in SystemC**. *3rd Mediterranean Conference on Embedded Computing (MECO)*, 2014, Budva, Montenegro.

- Iban Ayestaran, Carlos F. Nicolas, Jon Perez, Asier Larrucea, Peter Puschner: **Modeling and Simulated Fault Injection for Time-Triggered Safety-Critical Embedded Systems**. *IEEE 17th International Symposium on Object/Component /Service-Oriented Real-Time Distributed Computing (ISORC)*, 2014, Reno, USA.

- Jon Perez, Roman Obermaisser, Carlos F. Nicolas, Iban Ayestaran: **Modeling Time-Triggered Real-Time Control Systems Using Executable Time-Triggered Model E-TTM and SystemC-AMS**. *International Journal of Computer Systems Science & Engineering, Volume 26*, 2011.

Curriculum Vitae

Iban Ayestaran Cipitria

Bidebieta auzoa 1A Aiestaran Enea, 2 ezk. – 20400 Tolosa, Gipuzkoa
Basque Country (Spain)
✉ ibanayes@hotmail.com • in iayestaran

Education

Vienna University of Technology (TU Wien) **Vienna, Austria**
PhD *2010–present*
"Simulated Fault Injection for Time-Triggered Safety-Critical Embedded Systems"

Universidad de Navarra **Donostia / San Sebastián, Spain**
Automatics and Industrial Electronics Engineering *2008–2010*

Euskal Herriko Unibertsitatea (UPV-EHU) **Donostia / San Sebastián, Spain**
B.Eng. in Industrial Electronics *2005–2008*

Professional Experience

IK4-Ikerlan Research Center **Arrasate-Mondragón, Spain**
Embedded Systems Group Researcher *2010-present*
Research focus on safety-critical embedded systems (IEC-61508, ...), verification & validation and Fault-Injection

IK4-Ikerlan Research Center **Arrasate-Mondragón, Spain**
Master Thesis *2010–2010*
"Theoretical and Experimental Study of New Modeling Mechanisms with SCADE that Guarantee the Dependability of Embedded Systems"