UML-Based Design Space Exploration, Fast Simulation and Static Analysis

Daniel Knorreck

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

TELECOM ParisTech

2011-TELP-00XX

EDITE de Paris

Doctorat ParisTech

T H È S E

pour obtenir le grade de docteur délivré par

Télécom ParisTech

Spécialité “ Electronique ”

présentée et soutenue publiquement par

Daniel Knorreck

26/10/2011

UML-Based Design Space Exploration, Fast Simulation and Static Analysis

Directeur de thèse : **Ludovic Apvrille**
Co-encadrement de la thèse : **Renaud Pacalet**

Jury

Mme. Laurence Pierre, Université Joseph Fourier, Grenoble, France	Présidente	
M. Lothar Thiele, Swiss Federal Institute of Technology, Zürich, Swisse	Rapporteurs	
M. Frédéric Mallet, Université Nice Sophia Antipolis, Sophia Antipolis, France		
Mme. Cécile Belleudy, Université Nice-Sophia Antipolis, Sophia Antipolis, France	Examinateurs	
M. Jean-Luc Danger, Télécom ParisTech, Paris, France		
M. Ludovic Apvrille, Télécom ParisTech, Sophia Antipolis, France	Directeurs	
M. Renaud Pacalet, Télécom ParisTech, Sophia Antipolis, France		

T H È S E

Télécom ParisTech
Grande école de l’Institut Télécom – membre fondateur de ParisTech
46, rue Barrault – 75634 Paris Cedex 13 – Tél. + 33 (0)1 45 81 77 77 – www.telecom-paristech.fr

UML-Based Design Space Exploration, Fast Simulation and Static Analysis

COMELEC, Institut Telecom, Télécom ParisTech

Daniel Knorreck

A thesis submitted for the degree of

Doctor of Philosophy from Télécom ParisTech

Reviewers:
Prof. Dr. Lothar Thiele, Swiss Federal Institute of Technology Zürich
Prof. Dr. Frédéric Mallet, Université Nice Sophia Antipolis, France

Supervisors:
Prof. Dr. Ludovic Apvrille, Télécom ParisTech
Prof. Renaud Pacalet, Télécom ParisTech, France

«Never write anything that does not give you great pleasure. Emotion is easily transferred from the writer to the reader.»

Joseph Joubert

To Carina

Acknowledgements

All along this work, I had the pleasure to meet inspiring people that encouraged me, gave me advice, shared their opinion with me, guided me or were simply there when I needed someone to talk to. Without their valuable assistance, I wouldn't have succeeded this thesis. As all these people helped me in their own special way, it would be unfair to put their names in an order. That is why I chose the following way to express my gratitude:

X	X	R	R	J	I	P	E	J	K	C	L	H	Z	F
X	L	I	G	A	D	M	C	L	I	K	V	E	T	H
M	A	A	E	B	E	L	L	E	U	D	Y	N	R	W
J	U	D	S	A	R	A	B	E	A	C	M	D	A	H
F	D	H	E	U	R	E	C	O	M	D	H	R	U	O
G	S	E	B	A	S	T	I	E	N	X	W	I	D	C
A	F	P	M	A	L	L	E	T	O	M	J	K	E	I
B	X	B	Q	R	R	L	P	I	E	R	R	E	L	N
R	Z	F	V	T	O	C	H	A	F	I	C	T	R	E
I	D	E	H	T	D	A	N	G	E	R	O	D	E	Q
E	X	R	A	L	E	X	A	N	D	R	E	Q	N	T
L	T	I	Q	L	U	D	O	V	I	C	H	X	A	C
Z	D	E	B	B	E	R	N	D	P	A	U	L	U	B
S	A	L	C	A	R	I	N	A	C	C	T	E	D	W
R	P	I	N	T	H	I	E	L	E	B	V	M	P	H

Abstract

Abstract: Design Space Exploration at system level is carried out early in the design flow of embedded systems and Systems-on-Chip. The objective is to identify a suitable hardware/software partitioning that complies to a given set of constraints regarding functionality, performance, silicon area, power consumption, etc. In early design stages, accurate system models, such as RTL models, may not yet be available. Moreover, the complexity of these models comes with the downside of being demanding and slow in verification. It is commonly agreed that the only remedy to that problem is abstraction, which triggered the advent of virtual platforms based on techniques like Transaction Level Modeling. Non-functional, *approximately timed* models go even further by abstracting data to its mere presence or absence and introducing symbolic instructions.

The DIPLODOCUS methodology and its related UML profile realize the aforementioned abstractions. It relies on the y-Chart approach, that treats functionality (called application) and its implementation (called architecture) in an orthogonal way. DIPLODOCUS' formal semantics paves the way for both simulation and formal verification, which has been shown prior to this work. This thesis proposes enhancements to the methodology that make it amenable to verification of functional and non-functional properties.

At first, we focus on the way functional properties are expressed. As verification of high level models is usually conducted with temporal logic, we suggest a more intuitive way, matching the abstraction level of the model to be verified. The graphical but formal language TEPE is the first contribution of this work. To achieve a high level of confidence in verification in a reasonable amount of time, the model needs to be executed in an efficient way. The second contribution consists of an execution semantics for DIPLODOCUS and a simulation strategy that leverages abstractions. The benefit is that a coarse granularity of the application model directly translates into an increase in simulation speed. As a third contribution, we present a trade-off between the limited coverage of simulation and the exhaustiveness of formal techniques. Especially for large models, the latter may be hampered by the state explosion problem. As a result of data abstraction, DIPLODOCUS application models embrace non-deterministic operators. Coverage-enhanced simulation aims at exploiting a subset or all valuations of the corresponding random variables. Therefore, the DIPLODOCUS model is statically analyzed

and information characterizing the significant state space of the application is propagated to the simulator.
Finally, we provide evidence for the applicability of contributions by means of a case study in the signal processing domain. It will be shown that common system properties easily translate into TEPE. Moreover fast simulation and coverage-enhanced simulation provide valuable insights that may assist the designer in configuring a Software Defined Radio platform.

Keywords: Embedded Systems, System-on-Chip, Modeling, UML, SysML, Abstraction, Simulation, Verification, Graphical Property Language, System Level, Coverage

Résumé: L'exploration de l'espace de conception au niveau système est effectuée tôt dans le flot de conception des systèmes embarqués et des systèmes sur puce. L'objectif est d'identifier un partitionnement matériel / logiciel approprié qui réponde à un ensemble de contraintes concernant la fonctionnalité, la performance, la surface de silicium, la consommation d'énergie, etc. Lors des étapes de conception précoces, des modèles de système précis, tels que des modèles RTL, peuvent être encore indisponibles. Par ailleurs, la complexité de ces modèles présente l'inconvénient d'être exigeant et lent dans la vérification. Il est communément admis que le seul remède à ce problème est l'abstraction, ce qui a engendré l'apparition de plates-formes virtuelles basées sur des techniques telles que la modélisation au niveau transactionnel. Étant non fonctionnels, les modèles *approximately timed* vont encore plus loin en faisant l'abstraction de données simplement selon leur présence ou absence et en introduisant des instructions symboliques.
La méthodologie DIPLODOCUS et son profil UML correspondant réalisent les abstractions susmentionnées. La méthodologie s'appuie sur l'approche en Y, qui traite des fonctionnalités (appelées application) et leur réalisation (appelée architecture) de manière orthogonale. La sémantique formelle de DIPLODOCUS ouvre conjointement la voie à la simulation et à la vérification formelle, ce qui a été démontré préalablement a ce travail. Cette thèse propose des améliorations à la méthodologie qui permettent la vérification des propriétés fonctionnelles et non fonctionnelles.
Au début, nous nous concentrons sur la façon dont les propriétés fonctionnelles sont exprimées. Puisque la vérification des modèles de haut niveau est habituellement réalisée avec la logique temporelle, nous suggérons une façon plus intuitive qui correspond au niveau d'abstraction du modèle qui doit être vérifié. Le langage graphique, mais formel nommé TEPE est la première contribution de ce travail. Pour atteindre un niveau élevé de confiance en vérification dans un délai raisonnable, le modèle doit être exécuté effi-

cacement. La deuxième contribution vise donc une sémantique d'exécution pour les modèles DIPLODOCUS et une stratégie de simulation qui s'appuie sur l'abstraction. L'avantage est qu'une granularité grossière du modèle d'application se traduit directement par une augmentation de la vitesse de simulation. Comme troisième contribution, nous présentons un compromis entre la couverture limitée de la simulation et l'exhaustivité des techniques formelles. Lorsqu'il s'agit de modèles complexes, l'exhaustivité peut être entravée par le problème d'explosion combinatoire. En raison de l'abstraction de données, les modèles d'application DIPLODOCUS comportent des opérateurs non-déterministes. La simulation à couverture élargie vise à exploiter un sous-ensemble, ou bien l'intégralité, des valeurs des variables aléatoires. Par conséquent, une analyse statique des modèles DIPLODOCUS est effectuée et les informations caractérisant la partie significative de l'espace d'état de l'application sont propagées au simulateur.
Enfin, nous fournissons des preuves de l'applicabilité des contributions par le biais d'une étude de cas dans le domaine du traitement du signal. Il sera démontré que les propriétés courantes se traduisent aisément en TEPE. Par ailleurs, la simulation rapide et sa couverture élargie fournissent des indications pertinentes qui sont susceptibles d'aider le développeur à configurer une plate-forme radio logicielle.

Mots clés: Systèmes embarqués, Systèmes sur Puce, Modelisation, UML, SysML, Abstraction, Simulation, Vérification, Langage de propriétés graphique, Niveau Système, Couverture

Contents

List of Figures

Chapter 1

Introduction

Embedded systems are electronic devices whose computing elements are completely encapsulated in the device they control [109]. As opposed to conventional general purpose computers, the range of tasks an embedded system should accomplish is clearly defined. Nowadays, complex embedded systems may be integrated on one single chip and are thus referred to as Systems-on-Chip (SoC). SoC comprise a set of communicating electronic components on the one hand and complex software programmable parts on the other hand. SoCs are highly heterogeneous in nature: digital, analog and mixed signal components may be interconnected to form complex systems ranging from mobile hand sets and set top boxes to automotive controllers and feedback control systems for rail cars. Due to recent advances in the field of semiconductor physics, higher integration densities are achieved so that a given piece of silicon accommodates more and more transistors.
To make use of the available resources, **the complexity of embedded systems and Systems-on-Chip has been rapidly increasing** [35; 49]. On the one hand, users are demanding products exhibiting sophisticated features that are reliable, easy to use and affordable. On the other hand, the gap increases between integration and designer efficiency due to inadequate tools and methodologies. In addition to the rising demand of functionality, time-to-market is an issue of great concern. Hence, developers are facing significant difficulties due to an exponentially raising complexity. It becomes more and more unlikely that an optimal design represents an intuitive solution, thus the experience of the designer may not lead him/her to optimal designs with respect to functional and non-functional requirements such as performance, size, energy consumption, reliability. A whole body of work, including this thesis, is concerned with answering the question how the increasing complexity can be dealt with.

Given a particular functionality and associated requirements, the **design space is considered to encompass functionally equivalent implementation alternatives** [40]. Being almost infinitely large at the very beginning of the design flow, the design space should be gradually reduced during the design process by refining the model of the system. The less accurate the specification, the more indeterminism the system model exhibits,

and the larger the design space is. Optimally, a refinement results in a design which optimizes a predefined weight function of requirements.
Although the procedure seems straight forward in theory, **pruning the design space is very difficult to achieve in practice**. Experienced designers tend to leverage prior knowledge and stick to favored designs, which have proved to be well suited for previous products. Thus minor changes on the architecture are applied to derive a new one. While design reuse is a powerful means to cut down development costs of similar products by reducing the design space, it is not that capable when it comes to finding a close to optimal solution for innovative products. Also, platform-oriented design only transfers the problem of pruning the design space from the end-product vendor to the platform supplier. Hence, the essence of the problem remains unchanged and tools that allow for assessing different implementation alternatives for the same functionality are essential.

The analysis of systems at low abstraction levels assures a high degree of accuracy but comes with the downside of being demanding and slow. State-of-the-art simulation techniques operating at RTL, instruction or transaction level are not appropriate for system-level design space exploration for two reasons:

- Only a very limited number of implementation alternatives can be examined due to the high modeling effort and extensive simulation runtime.
- The lack of specification at early design stages may prohibit the construction of detailed models - even if the effort was acceptable.

Thus, the use of **abstractions is unavoidable** [35] when performing System Level Design and should be part of a thoroughly defined modeling methodology. The use of abstractions implies as well that application and architecture concerns are handled in an orthogonal fashion. Indeed, for the sake of reusability, an application model should not need to be rewritten when being experimented on different platforms. This policy is known as the Y-Chart approach [77] and widely used in the landscape of System Level Design Space exploration.

1.1 Problem Statement

The work presented in the scope of this thesis advocates **techniques to alleviate design tasks at early design stages**. In that context, the previous section has already pointed out the need for abstractions. We will now have a closer look at the two main types of abstraction, yielding either the functional aspect or the timing aspect of a system.
Consequently, two orthogonal views on the system intended for design are prevalent. On the one hand, the designer relies on purely functional, untimed models to examine especially intricate algorithmic parts of the application. For example, if the suitability of a Quadrature amplitude modulation (QAM) decoder is to be looked into, the expected

outcome of the analysis is whether or not the decoder is able to reconstruct the original sequence of samples for arbitrary signals. The availability of the samples at the right time is taken for granted as the provisioning of the decoder with data is abstracted away. Functional correctness is the only concern at this stage. In the domain of signal processing and control engineering, mathematical tools such as Matlab have proven their value for fast prototyping.

On the other hand, it is of utmost importance to have a global view of the interplay of system components. Thereby, attention is consequently drawn to timing and performance allowing functionality to be extensively abstracted. As a consequence, the bird's eye view on a system aids the developer to arrange and dimension components and communication infrastructure in a way that non-functional constraints are met. To get back to the QAM decoder example, the goal would be to figure out which implementations guarantee that the decoder never runs out of input samples and that the output samples arrive on time at their destination. In that case, the internal computations can be abstracted to symbolic instructions as merely the I/O behavior is of interest in this analysis. This directly leads to the notion of non-functional performance models this work is mainly concerned with.

The contributions of this thesis were made in the context of the DIPLODOCUS framework, embracing a UML profile, a methodology and related tooling. It is especially suited for reasoning about abstract (in terms of functionality), control dominated performance models of today's SoC. The framework complements the initially presented fully functional, but untimed models. Also in domains traditionally relying on untimed models, the interplay of various (signal) processing routines is getting more and more sophisticated and needs to be orchestrated by control-centric algorithms. Despite this increasing need for an adequate tooling, state-of-the-art environments of academia and industry are often hampered by the following shortcomings:

- Application (functionality) and architecture (platform) issues are not handled in an orthogonal fashion [60] to speed up HW/SW partitioning.
- Emphasis is exclusively put on either simulation or on formal methods.
- A trade-off between these two extreme cases of verification is not provided.
- Abstractions are not fully leveraged to perform fast simulation.
- Performance is the only concern; control flow cannot be modeled/verified.
- The methodology does not feature a modeling standard that enforces abstractions (such as UML).
- Details of underlying algorithms must be provided in the form of source code in order to execute the model.

- After model transformation to an executable counterpart, debug information is not back-propagated to the original model.
- The level of abstraction of the language used to express functional properties does not match the level of abstraction of the system model (Example: system model in UML, verification language is CTL).

Some of the aforementioned downsides have been remedied in the context of research prior to this thesis. In the next section, the objectives of this work are surveyed.

1.2 Objectives and Contributions

This thesis is devoted to the enhancement of an existing Design Space Exploration environment which is introduced in Chapter 2. Application functionality and architecture are modeled by means of the previously introduced UML profile DIPLODOCUS. The latter is very capable when it comes to modeling complex systems as it introduces data and functional abstractions. DIPLODOCUS is supported by the open source toolkit TTool that, prior to this work, was equipped with modeling features to draw diagrams, a rudimentary simulation engine, and an automated model transformation to the formal languages LOTOS and UPPAAL. The main contributions were made in the field of simulation, coverage enhancement of models, expression of functional properties and in the optimization of the design flow:

- An efficient **simulation and validation strategy** was conceived to complement the formal capabilities of the framework. The novel simulation algorithm takes heavily advantage of the properties of the application model with regards to granularity and abstractions. An execution semantics for DIPLODOCUS operators has been defined which is leveraged in simulation and matches the abstractions inherent to the profile.
- Attention was devoted to finding a **compromise between the limited coverage of conventional simulation and exhaustive formal verification**. To extend the coverage of simulation, an algorithm is proposed which statically analyzes DIPLODOCUS applications and identifies the set of significant state variables at a given point in the application. Methods are presented to exploit results of the static analysis during simulation with the objective to examine several execution branches. In case recurring system states are encountered, simulation of particular branches may be abandoned. With respect to conventional model checking techniques, the coverage of the application model can be varied and constraints of the architecture are taken into account. This considerably limits the state space explosion problem which is encountered when model checking application models alone.

- Verification is often hampered by the obligation to rely on completely different languages than those used for system modeling. For example, **verification of a UML system model should be feasible within the same environment** in UML. To address this issue, the verification language TEPE is introduced, a graphical TEmporal Property Expression language based on SysML parametric diagrams. TEPE enriches the expressiveness of other well-established property languages in particular with the notion of physical time, easy to express logical and sequential properties and highly composable operators. Thanks to two dimensional composition, TEPE supports both events and states based formalisms . Besides, TEPE is endowed with a formal semantics which is also part of the contributions of this work.

- The design flow has been optimized in the sense that the user does **not need to refer to the executable model for debugging purposes**. Construction, debugging, and verification can henceforth be seamlessly accomplished in the same environment, using the same language, without having to write a single line of code.

- Last but not least, this thesis also comprises an extensive practical part. A prototype has been developed which **showcases the above mentioned concepts**.

1.3 Outline

This remainder of this thesis is structured in 8 main chapters:

- Chapter 2 puts this thesis into the context of research carried out at our laboratory. The objective is to make the reader familiar with the DIPLODOCUS framework so as to ease the understanding of subsequent chapters. Furthermore, a clear boundary is drawn between existing elements of the framework prior to this thesis, and enhancements being part of the contributions.

- Chapter 3 positions the contributions in the landscape of related work on frameworks for System Level Performance Analysis attempts to speed up simulation, verification languages as well as visualization capabilities of development environments. A classification and a thorough analysisof a considerable body of work motivates efforts and contributions subsequent chapter elaborate on. Finally, an overview of widely used Models Of Computation (MoC) is given to have a framework for presenting the DIPLODOCUS MoC in 5.4.

- Chapter 4 presents the verification language TEPE, both intuitively and formally and justifies our decision to build the language upon SysML parametric diagrams. It surveys the SysML AVATAR profile for embedded systems, in the context of which TEPE was initially developed. After exemplifying the use of TEPE with some properties, light is shed on the integration of TEPE into DIPLODOCUS. At

the end of Chapter 4, 5 and 6, the interested reader may get some practical insights into implementation issues in the corresponding sections.

- Chapter 5 covers the simulation semantics of DIPLODOCUS, the algorithm used to simulate DIPLODOCUS models and it elaborates on design decisions such as the use of C++ instead of SystemC. Furthermore, information on the automated model transformation of a UML model to its C++ counterpart are provided.
- Chapter 6 exposes the methodology to enhance coverage of conventional simulation based on static analysis and model checking techniques.
- Chapter 7 is concerned with practical matters regarding the Tooling which are however of importance to make the theoretical contributions applicable in practice.
- Chapter 8 discusses a case study performed in the field of digital signal processing with the objective to demonstrate the applicability of the concepts.
- Chapter 9 concludes the thesis with an outlook on future research and finally summarizes the contributions and publications.

Chapter 2

The DIPLODOCUS environment for Design Space Exploration

2.1 Introduction

This Chapter defines the context of this work, which is System Level Design Space Exploration and introduces the methodology developed at our laboratory. As stated in the previous chapter, the contributions highlighted in Chapters 4, 5, 6, 7 were made within the modeling framework called DIPLODOCUS [14; 129]. Section 2.6 clearly distinguishes contributions from existing elements of the framework prior to this thesis. DIPLODOCUS is a UML profile targeting the design of System-on-Chip at a high level of abstraction. A profile customizes UML [91] for a given domain, using UML extension capabilities. This practice has the major advantage that the introduced language does not have to reinvent the wheel in terms of syntax. Instead, predefined UML primitives are reused and are assigned a user defined semantics. Moreover, a UML profile is usually accompanied by a methodology so as to guide the user in his modeling efforts. Recent history has shown that the acceptance of profiles hinges with adequate tool support. Therefore, DIPLODOCUS is supported by a dedicated toolkit called TTool [16]. Section 2.4 justifies the usage of DIPLODOCUS with respect to the MARTE [93] profile, which is increasingly gaining attention both from users and tool vendors. Section 2.5 gives some insight into approaches to fine-tune high level models to improve their accuracy.

2.2 Design Space Exploration

Design Space Exploration (DSE) is the process of assessing viable implementation alternatives providing a defined set of functions with respect to non-functional metrics such as performance, area, power consumption, heat dissipation, etc. By definition, only solutions complying to functional constraints constitute the design space. That means DSE is linked to the process of further constraining a correct and properly defined set of functions by for example binding them to particular processing units. Throughout

the following chapters, we will be concerned with System Level DSE also referred to as hardware software partitioning (the decision whether to implement a function in hardware or in software). However, it should be emphasized that DSE is of course not limited to that specific domain. DSE is similarly applied in other contexts and at different stages of the design flow of SoCs:

- Evaluation of several pipeline based architectures for Application Specific Integrated Processors (ASIPs), most efficiently carried out by means of architecture description languages such as LISA [10; 130]
- Analysis of memory hierarchies and caches in particular to match the demand for data accesses [99; 118]
- Assessment of bus and interconnect architectures in general with the objective to minimize contention [61; 72; 73; 94; 133]
- Trading off delay and area in logic synthesis

Indeed, DSE originates from logic synthesis where the performance of a circuit improves at the expense of an increased silicon area [40], provided that functions exhibit a sufficient degree of parallelism. In general, the design parameters span the design space and a variation of design parameters is tantamount to moving through this space. An evaluation of a single design point reveals whether the given requirements are met or the space has to be explored further. Obviously, the whole design space has to be explored in general to spot optimal solutions if no specific knowledge of the problem domain is available. In case an exhaustive search is too costly in terms of execution time or memory consumption, heuristics may be incorporated to prune the design space. Often, covering the whole design space is not feasible due to its sheer size. Nowadays, algorithms are capable of leveraging the knowledge of encountered design points in order to direct the search in an intelligent way. Pareto optimal algorithms for instance yield design points where an improvement of an objective inevitably leads to a degradation of another. However, an automated exploration of the design space is often restricted to a particular family of architectures or is otherwise constrained.

When exploration tasks are performed at system level, abstractions on either data or control flow are necessary to keep complexity within reasonable limits. The Y-Chart approach [60] enables the designer to address separately functionality, architecture and mapping issues. Applications are defined according to an underlying model of computation which captures important properties with respect to a specific problem domain. Architecture models mimic a concrete hardware component or rather stand for a whole family of components. Generic components normally expose parameters to the user which have to be well configured in order to model the desired behavior. A mapping stage binds functional entities to architecture building blocks. The subsequent evaluation stage may require model transformations (synthesis steps, compilations, code

adaptations) of the inputs models (Kahn Process Networks, Activity Diagrams,...) to executable representations (for example C++, SystemC code). After having carried out simulation, formal verification or static analysis, the designer has to validate the results against the requirements. Further iterations on the involved models may be necessary to obtain acceptable results.

2.3 Methodology

The UML-based DIPLODOCUS methodology for system Design Space Exploration (DSE) is depicted in Figure 2.1. It obliges the user to adhere to the Y-Chart approach [60] which has been extensively discussed in literature. In short it states that if application (functional view) and architecture (platform view) are handled in an orthogonal fashion, the burden of experimenting with several design points is considerably alleviated. The Y-Chart approach is thus very capable when it comes to DSE, where an application is to be assessed with different architecture constraints.
The design flow embraces the following three steps:

1. **Applications** are first described as a network of abstract communicating tasks using a UML class diagram or a component diagram. This is the static view of the application. One behavioral description per task must be supplied in terms of a UML Activity Diagram.

2. Targeted **architectures** are modeled independently from applications as a set of interconnected generic hardware nodes. A set of parameters permits to calibrate components for their application area. UML nodes were defined to model HW elements (e.g. CPUs, buses, memories, hardware accelerators, bridges).

3. A **mapping** process defines how application tasks are bound to execution entities and similarly how abstract communication channels between tasks are bound to communication and storage devices.

Within a SoC design flow, DSE is supposed to be carried out at a very early stage. Hence, the main DIPLODOCUS objective is to help designers to spot a suitable hardware architecture even if algorithmic details have not yet been stipulated thoroughly. To achieve this, DIPLODOCUS relies (i) on simulation and formal proof techniques, both at application and mapping level, and (ii) on application models clearly separated from architecture models. Depending on the respective algorithm, simulation speed may benefit from the high abstraction level of both application and architecture models, as compared to simulations usually performed at lower abstraction levels (e.g. TLM level, RTL level, etc.). Additionally, formal analysis techniques may be applied before and after mapping. So far, DSE is not automated in the sense that simulation results and constraints drive modifications on the architecture. However, there have been various efforts in that field [25; 42] which could successfully be leveraged for that purpose.

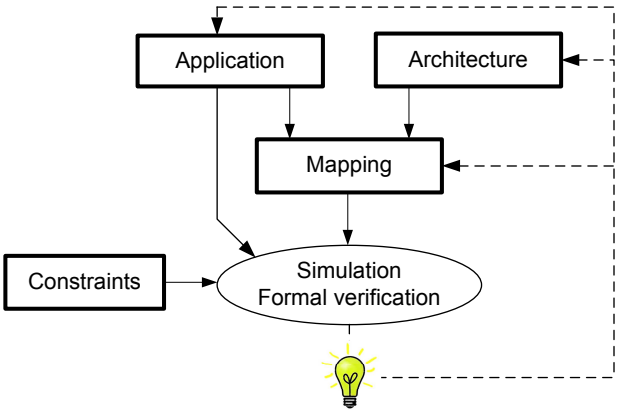


Figure 2.1: The Y-Chart approach and DIPLODOCUS

2.3.1 Application model

An application model is the description of functions to be performed by the targeted SoC. Data-dependent decisions are abstracted by means of indeterministic operators. An application model may give rise to several execution traces due to its inherent indeterminism and the potentially undefined partial order of concurrent actions. At application modeling level, computations and communication transactions are accounted for by abstract cost operators. The time it takes to process the latter can only be resolved with the aid of parameters annotated to the architecture model. Abstract cost operators entail two kinds of abstractions which reflect the degree of uncertainty inherent to early design stages:

- **Data abstraction**: Only the amount of transferred data is taken into account, not the data itself.
- **Functional abstraction**: Algorithmic details are abstracted by means of their symbolic cost operators.

As stated before, functions are modeled as a set of abstract tasks described within UML class diagrams. Task behavior is expressed using UML activity diagrams that are built upon the following operators: control flow and variable manipulation operators (loops, tests, assignments, etc.), communication operators (reading/writing abstract data samples in channels, sending/receiving events and requests), and computational cost operators. This section briefly describes a subset of the aforementioned operators as well as their semantics and provides definitions for Channels, Events and Requests:

- **Channels** are characterized by a point-to-point unidirectional communication between two tasks. Three Channel types exist:
  - Blocking Read/Blocking Write (BR-BW)

  - Blocking Read/Non Blocking Write (BR-NBW)
  - Non Blocking Read/Non Blocking Write (NBR-NBW)

- **Events** are characterized by a point-to-point unidirectional asynchronous communication between two tasks. Events are stored in an intermediate FIFO queue which may be finite or infinite. In case of an infinite queue, incoming events are never lost. When adding an event to a finite FIFO, the incoming event may be discarded or the event that arrived earliest may be dropped if the FIFO is full. Thus, a single element queue may be used to model hardware interrupts. In tasks, events can be sent (Send Event), received (Wait Event) and tested for their presence in a queue (Notified).
- **Requests** are characterized by a multi-point to one point unidirectional asynchronous communication between tasks. A unique infinite FIFO between senders and the receiver is used to store all incoming requests. Consequently, a request cannot be lost.

As some of the constituting operators of Activity Diagrams are referred to in later sections, a brief survey is provided which is not meant to be exhaustive (see Table 2.1).

2.3.2 Architecture model

A DIPLODOCUS architecture is built upon the following parameterized hardware nodes:

- **Computation nodes**: Typically, an abstract CPU model merges both the functionality of the hardware component and its operating system. The behavior of a CPU model can be customized by the following parameters (amongst others): data size, pipeline size, cache miss ratio, number of cores and scheduling algorithm.
- **Communication nodes**: A communication node is either a bus or a bridge. The bus model exhibits the following parameters: data size, latency, number of channels and scheduling policy. Note that connectors established during the mapping stage are supposed to interconnect a hardware node - except for buses - with a bus. A connector may be annotated by a priority if the respective bus has a priority-based scheduling policy.
- **Storage nodes**: Memories are parameterized with two measures: latency and data size.
- **DMAs**: So far, DMAs are represented with an adequately parameterized CPUs and a dedicated task mapped onto it.

A DIPLODOCUS architecture is modeled in terms of a UML deployment diagram where DIPLODOCUS links and nodes have their own UML stereotype.

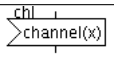
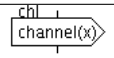
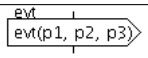
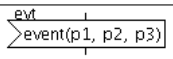
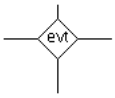
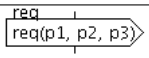
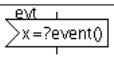
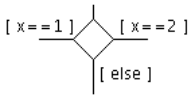
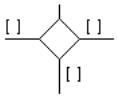
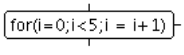
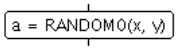
Icon:	Command:	Semantics:
	Start	Denotes the beginning of a task
	End	Denotes the end of a task
	Read Channel	Read x samples from a channel
	Write Channel	Write x samples into a channel
	Send Event	Send an event including 3 parameters
	Wait Event	Wait for an event carrying 3 parameters
	Select Event	Wait for one of several events
	Request	Make a given task runnable
	Notified	Return the number of pending events
	Execi	Perform x computations
	Execi Interval x y	Perform between x and y computations
	Choice	Deterministic choice
	Random Choice	Indeterministic choice
	Sequence	Execute connected branches consecutively
	Loop	For loop construct
	Action	Variable assignments
	Random	Determine a random number between x and y

Table 2.1: Most prevalent DIPLODOCUS operators for task behavior

2.3.3 Mapping

A DIPLODOCUS mapping describes the association of application elements - i.e. tasks, channels, requests and events - and hardware nodes. Thereby the following rules apply:

- Abstract tasks must be mapped on exactly one computation node. This is not restrictive as the node may support concurrency (e.g. multi-core CPUs).
- Abstract communication entities must be mapped on communication and storage nodes. A channel is usually mapped on n buses, n-1 bridges and exactly one storage element. Furthermore, all connected communication links have to form a continuous path without loops.

Depending on the mapping semantics, additional parameters may be of interest. For example, when mapping a task on a CPU node having a priority-based scheduling policy, task priorities have to be defined.
The mapping stage is carried out based on previously created DIPLODOCUS architecture diagrams: artifacts representing tasks and channels are simply bound to hardware components in a drag and drop fashion. Post-mapping specifications contain less traces than pre-mapping specifications since a mapping is intended to resolve shared resource allocations. In contrast, the application model alone does not stipulate any temporal order of concurrent actions, apart from causality constraints due to synchronization. Moreover, traces obtained after mapping are supposed to be a subset of traces obtained before mapping.

2.3.4 Nomenclature

Several notions are used abundantly throughout this thesis and therefore need some closer attention. A formal definition of DIPLODOCUS operators and related notions is out of scope of this work and can be found in [55].

- **Command** and **Operator** are used as synonyms and refer to the building blocks of an Activity Diagram. They describe the behavior of DIPLODOCUS tasks. The operators relevant for this work are depicted in Table 2.1.
- The execution semantics of DIPLODOCUS tasks implies that commands are not considered as atomic. Commands can be split into smaller portions to satisfy application and architecture semantics. This portion is henceforth referred to as **Transaction**. As an example, an Execi 10 command may be broken down into two transactions of length 8 and 2 respectively.
- The separation of concerns demands for different units of measurement for complexity and time. At application level, complexity is specified in terms of Execi units or data samples and is called **(virtual) length**. As soon as a transaction is bound to HW devices, its physical duration may be computed as a function of its

(virtual) length and device parameters. The latter measure is therefore referred to as **duration**.

- The notion of "Channel" appears in two contexts in this work: first a Channel denotes a means of communication between two DIPLODOCUS tasks. Second, if a bus may handle several data transfers concurrently, it is said to provide several channels. The intended meaning should become clear from the context.

- Unfortunately, the notion of **Event** is overly stressed in different communities. First, in DIPLODOCUS it stands for a means of synchronization among tasks. Second, it is used in a broader sense to denote a transition from one system state to another. The latter meaning applies in the context of Models of Computation (cf. Section 3.2) and the simulation kernel (cf. Section 5.6.3).

- In DIPLODOCUS, behavior may be captured with a class diagram or likewise with a component diagram. **Classes** and **components** share the same semantics; they are defined as concurrent functional entities with their own control flow. In the following, both notions are referred to as **tasks** to abstract from the underlying diagram type.

2.4 A word on MARTE

Having gained an insight into DIPLODOCUS, the reader may ask the legitimate question why we did not rely on the MARTE [93] profile. MARTE constitutes an extension of the UML standard and its major concerns are specification, design, verification and validation stages of real time and embedded system development. Packages are devoted to resource modeling, non-functional properties, value constraints, hardware/software resource modeling, allocation modeling and schedulability analysis. The profile thus replaces the UML Profile for Schedulability, Performance and Time. MARTE seeks to establish a common ground for reasoning about systems, by standardizing a particular syntax. This policy enables designer to exchange models internally and between third parties in a standard format. However, MARTE does not stipulate any particular semantics or methodology; this is left to the user or a methodology provider respectively. MARTE allows to annotate models with additional information which can be leveraged for performance and schedulability analysis.
Even if the contributions of this thesis are exemplified with the aid of the DIPLODOCUS framework, they apply just as well to any other language exhibiting the same semantics, be it a textual form, MARTE, or others. The syntactical structure of the language has no impact on verification, simulation or coverage procedures presented herein. Nothing prevents the user from modeling the application in terms of MARTE blocks stereotyped as schedulable resources, to make use of the MARTE Hardware Resource Model in architecture diagrams, to bind schedulers (defined with the MARTE GaExecHost stereotype) to components using the MARTE allocate stereotype, etc.

In any case, MARTE Activity Diagrams would have to be enriched with the operators described in 2.3.1.

2.5 Model calibration

Regardless of the formalism or UML profile used, the major goal of high level models is to minimize the modeling effort and to reach a high execution speed. However, the obtained performance figures will not lead the designer to the right design decisions if they are too far from the actual system behavior. Therefore, an issue of great concern is to yield sufficiently insightful performance estimations despite the applied abstractions. To achieve this, the parameters exposed by abstract models have to be fine-tuned best possible to the respective circumstances. In that context, two use cases can be discriminated:

- The system itself or a similar one has already been refined so that low level models (source code for software, RTL for hardware) are available. High level modeling in that case constitutes an efficient way to extend the scope of the analysis from component to system level and to improve simulation speed significantly.
- The system under design is novel so that no suitable low level model is at hand. High level modeling is the only way to get some first insights into the performance.

In literature [70; 103], four approaches for calibration are prevalent:

1. Deducing the number of operations from a purely algorithmic description: therefore, from a representation in pseudo-code, Matlab, or other high level domain specific languages the number of instructions (different types of operations, memory accesses, etc.) is calculated
2. Extracting data from measurements or traces of similar existing systems: the first step would consist in separating the individual contributions to system load, i.e. the different applications. Thereafter, the utilization of a CPU would be measured separately for each application and the results merged as a function of the particular use-case. The last step aims at differentiating memory accesses from processing instructions based on their assumed probability of occurrence.
3. Inferring the workload from low level models, for instance source code: The simplest approach would just entail cross-compiling an application for the target platform and counting execution and memory access instructions. However, provided that the high level model reflects control flow like DIPLODOCUS, the challenge to match assembly instructions with high level control structures has to be dealt with. In the presence of sophisticated optimization features of today's compilers, it is obvious that this is not a trivial exercise (cf. [116; 128]).

4. Rather than relying on tabulated values for calibration, so called on-line calibration could dynamically compute the exact values at simulation run time. This is tantamount to integrating models of heterogeneous abstraction levels in order to trade off simulation performance against increased accuracy. As this method is currently not foreseen in DIPLODOCUS, it is not elaborated in further detail.

All hardware dependent measures obtained with the aforementioned strategies would of course be annotated to the DIPLODOCUS architecture model. To make sure platform-independence is respected, the DIPLODOCUS application model merely comprises figures characterizing the complexity of an algorithm in terms of instructions of particular type (floating point instruction, FFT, etc.). In the further course of this thesis, the developer is assumed to have an adequate strategy for model calibration at hand.
In the following, several questions regarding model calibration and best practices of DIPLODOCUS modeling will be answered in an FAQ manner.

Data abstraction: boon or bane?
It should be reemphasized that data-dependent behavior of the application has to be expressed in terms of random operators. That is, a stochastic model of data hazards has to be embedded into the application model. However, this effort is not particular to our methodology, neither it is to a high level of abstraction. Whenever a system is loaded with data-dependent tasks, the designer is obliged to come up with a statistical model of the data to be processed. Only with that model, it is possible to avoid overdimensioning the system for the worst case. The statistical model gives the designer the confidence that an architectural trade-off delivers an acceptable performance with a known probability.

Which granularity has to be applied for data exchanges (Read/Write operators for channels)?
Basically, all commands manipulating data could in theory entail a Read/Write transaction on a bus. Even if one considers a simple command such as an assignment, it is not guaranteed that the variables referred to are held in cache. One can discern two main types of data: on the one hand data related to control flow (loop variables, flags, ...) and on the other hand input or output data of an algorithm. The fast discrete cosine transform algorithm for example exhibits two consecutive loops containing the computation of frequency coefficients as a function of the given input samples. Control flow merely relies on the loop variables. Fortunately, the usage of control flow variables often satisfies the temporal and the locality criterion of data caches. Furthermore, the amount of control data is often neglectable as compared to the input and output values. For these reasons, control data transfer is not modeled with DIPLODOCUS channel operators. The model of the DCT algorithm is thus reduced to reading the input samples, executing the main algorithm and writing back the output frequencies to memory.

What about the detailedness of an algorithm?
The algorithmic model should cover branches that differ significantly in terms of processing time. This means that the model should not reflect *if* statements spanning only a few cycles. In this case, an average value of cycles should be applied in favor of simulation speed (an Execi Interval operator could be used as well). Adequate statistical models should be provided which guarantee a realistic behavior in spite of the lack of "real input data". DIPLODOCUS Action and Choice commands can be used for the necessary computations of Exec units and to direct the control flow.

Which granularity of partial order to choose for read, execute and write operators?
The most simplistic model of an application would just consist of a read (for input data), an Execi operator and a Write operator (for output data). The other extreme case would be to spread the three operations more or less uniformly across the task, enriched with control flow operators. The simulation algorithm presented in this thesis (Chapter 5) is able to take advantage of long transactions in order to process more clock cycles per time unit. Thus, simulation performance benefits from a coarse grained model to the detriment of simulation accuracy. Consequently, there is no silver bullet to address this problem, as its solution highly depends on the particular simulation objective.

2.6 Putting Contributions into context

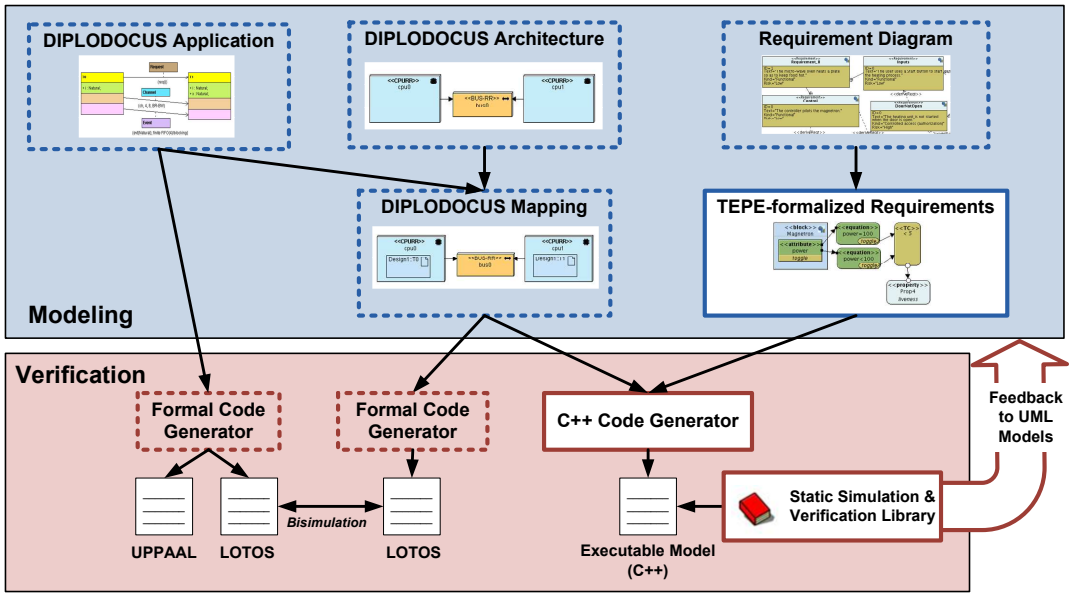


Figure 2.2: DIPLODOCUS design flow

The open-source toolkit *TTool* [16] supports several UML2 / SysML profiles, in particular DIPLODOCUS [11], AVATAR [65], TURTLE [13], CTTool [8] and Network Calculus [15].

TTool offers UML model editing facilities as well as press-button approaches for formal verification. TTool and its profiles have proven their value in several projects conducted with academic and industrial partners.
Formal verification can be conducted on the application model alone or on the mapped model, embracing an application and an architecture (cf. "Formal Code Generator" labels in Figure 2.2). TTool is interfaced to verification tools by means of the intermediate formal languages LOTOS and UPPAAL. However, these languages are transparent to the user thanks to automated model transformations. TTool offers a user-friendly interface to check simple properties such as liveness and reachability (e.g., with UPPAAL) of UML operators. A reachability graph may also be transformed into a Labeled Transition System, a structure for which CADP [38] implements minimization techniques based on trace or observational equivalences just to mention a few. Formal techniques may further be leveraged to carry out formal Design Space Exploration of Systems-On-Chip [62].
In Figure 2.2, the achievements of this work are marked with a white background and continuous borders, while existing elements are marked with a dashed border. Prior to this thesis, merely liveness and reachability of DIPLODOCUS operators could be checked at the push of a button. To this end, the UPPAAL verifier was invoked with the corresponding CTL formula. When it came to more complicated and user defined properties, they could only be expressed by means of SysML requirement diagrams in an informal way, namely in plain text. The only way to formalize them was to provide CTL formulas to TTool and which were simply forwarded to the model checker. This work eliminates this rupture of the design flow by proposing a UML-based, formal verification language suited for the abstraction level of DIPLODOCUS (see Chapter 4). Moreover, a fast simulation engine was developed [63] which provides visual feed-back to UML models [64] (see Chapter 7) and thus allows for debugging and verification directly in the source model. Chapter 5 elaborates on the simulation algorithm, whereas pivotal techniques permitting the extension of the simulation coverage are discussed in Chapter 6.

2.7 Conclusions

In summary, this chapter made the reader familiar with the high level Design Space Exploration Environment DIPLODOCUS, the homonymous profile, its methodology and operators. It is based on the Y-Chart approach acknowledging the need for separate models for application and architecture. That way, the developer may experiment very efficiently with several possible implementations for a given set of functions. It has been stated that the profile MARTE could also be used to express the semantics of DIPLODOCUS, if Activity Diagrams were enriched with some additional operators. The contributions of this thesis are however not impacted by the concrete syntax of the modeling language, as long as the semantics is respected. Lastly, this chapter dealt with the crucial issue of parameterizing high level models so as to improve their accuracy.

Algorithmic descriptions, low level model and knowledge of the field of application may be leveraged for that purpose.
After the existing environment and the intended enhancements have been presented, the next chapter justifies the latter by examining related work in the field of system level modeling, simulation and verification of Systems-on-Chip.

Chapter 3

Approaches for System Level DSE and Verification

3.1 Introduction

This Chapter surveys efforts to verify non-functional and functional properties of Systems-on-Chip (SoC) early in their design flow. Section 3.2 paves the way for the analysis of related work and the DIPLODOCUS simulation semantics (section 5.4). The section covers the common ground of all modeling efforts, namely Models of Computation (MoCs). Section 3.4 focuses on System Level Design Space Exploration (DSE) where models are tailored towards performance aspects and are not necessarily functional. In that context, approaches can be roughly classified into three categories, to which separate sections are devoted: formal/static approaches (Section 3.4.1), simulation centric approaches (Section 3.4.3) and hybrid variants (Section 3.4.4). For a more thorough classification of approaches please refer to section 3.3. Formal and simulation based environments are contrasted with each other as well as approaches targeting whole systems or solely communication architectures (Section 3.4.5). Depending on their methodology and the underlying MoC, frameworks may be restricted to a particular range of applications (such as signal processing applications) and attain a different coverage of the design space. Simulation as a verification method plays a pivotal role in this thesis. To that end, methods aiming at speeding up traditional simulation techniques are surveyed as well in Section 3.4.6. Section 3.5 justifies our property expression language TEPE with respect to popular approaches in that domain. To acknowledge the importance of tooling for the acceptance of a methodology, TTool is related to state of the art UML modeling environments embracing visualization features (see Section 3.6).

3.2 Models of Computation

As this section demonstrates, Models of Computation play a pivotal role in system modeling and verification. In later sections, related approaches for DSE are compared

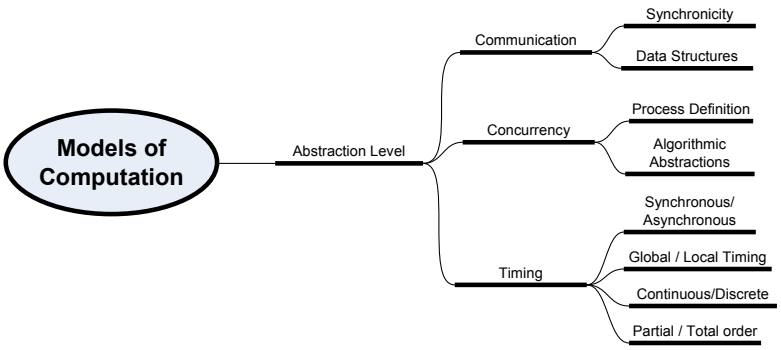


Figure 3.1: Classification of Models of Computation

with respect to the underlying MoC (amongst other criteria such as the capabilities of verification features). For this reason, this section paves the way for the analysis of related work by providing definitions and examples of MoCs. Moreover, general properties of MoCs are related to the DIPLODOCUS profile in particular. DIPLODOCUS semantics of application models was formally defined and justified prior to this work in [11; 14]. However, the semantics of hardware components and mapping leaves room for different interpretations. Section 5.4 intents to clarify simulation semantics and justifies the decisions.

Engineering as a discipline is heavily tied to the process of modeling, which refers to solely taking into account relevant properties and parts of a system under design. Details that do not impact design decisions at the current stage can be abstracted away (compare Section 3.4.3.1). The art of modeling amounts to discriminating the relevant from the irrelevant with respect to the expected outcomes, be it performance figures, compliance to properties, etc.
As soon as digital electronic systems come into play, basic atomic operations of the system are referred to as computations.

In DIPLODOCUS . . .
. . . data abstraction, indeterministic operators and symbolic operations attempt to abstract system behavior while preserving performance relevant properties to a large extent.

. . . transactions of length one (1 Execi unit, 1 data sample) are considered as atomic actions.

A MoC specifies the set of allowable operations and may account for their respective costs, the state change provoked in the system, or both. Thereby, MoCs may abstract from the underlying physical details to capture only features relevant to the problem.

. . . there are mainly two allowable operations: control flow related ones taking no time (Choice, Action, etc.), and operations with which are associated a cost (Execi, Write, Read, etc.)

This suggests that the choice of an adequate MoC highly depends on the nature of the application to be modeled and the aspects to be investigated. For instance, control flow issues of a video player application could be expressed in terms of Petri nets, whereas the signal processing part is analyzed by means of synchronous dataflow models.

. . . emphasis is placed on performance modeling. However, ongoing research investigates to what extent and under which conditions functional properties are preserved with respect to more detailed models. DIPLODOCUS is control flow centric and therefore less suited for modeling data flow systems which are always assumed to be in a (single) steady state.

According to [35] and [74], a MoC can be characterized by the following elements:
An **event** comprising a tag (for example the time of occurrence) and a value (for example an ID denoting the nature of the event). Timed MoCs impose the order of tags, untimed MoCs may merely establish a causal/partial order. Events often trigger the transition to a new system state.

*. . . application models are untimed and thus merely impose a causal order on **events** (not to be confused with DIPLODOCUS events). Only after the mapping stage the timing of events can be resolved.*

A **process** is considered as a set of possible behaviors relating input signals to output signals. The MoC determines the character and the building blocks of the processes the application is made of.

*. . . the notion of"task" corresponds to a **"process"** in the MoC terminology.*

The **state** of a system makes the system's history irrelevant or in other words the system cannot distinguish between histories which lead to the same state. The state of the system embraces all pieces of information needed to determine the responses to future inputs.

*. . . the **state** space is spanned by local variables of tasks, the current position within a task and the state of all synchronization primitives. The architecture can be seen as an instrument to limit the partial order of the application, which does not necessarily extend the state space.*

A MoC is chosen with regards to the investigated properties: formal proofs certainly require a solid mathematical foundation of the MoC, whereas executable MoCs are more tailored towards simulation. In general, SoC design should be based on two models. On the one hand, a logical mathematical model involves nice properties which allow for performing formal proofs (for instance by means of static analysis, model checking,...).

... models may be verified by both simulation and formal proofs thanks to its formal semantics [62].

On the other hand, the limitation of these models is that aspects contributing to the system's physical behavior are not well captured. The underlying assumption is that computation power of the computation engine is abundant and physical limitations do not have any effect. The space of expressible designs is reduced in favor of stronger and more sound mathematical underpinning [109]. On the other hand, this assumption is often too strong in the field of embedded systems and therefore more programmatic and less formal models complement the tool box of the designer. The latter models focus more on physical aspects of the system, like for instance the Discrete Event MoC, which is the underlying MoC for several hardware description languages.

... the separation of concerns leaves the choice to the user whether physical constraints are taken into account.

It is obvious that when dealing with complex Systems-on-Chip, there is no omnipotent MoC which is capable of expressing all system properties of interest. Therefore, designer often resort to several of them.

... data produced by algorithms is abstracted and therefore cannot be verified. To this end, verification should be complemented with purely functional, untimed models written in domain specific languages such as Matlab.

MoCs are classified [82] according to three orthogonal axes (depicted in Figure 3.1): Time describing the evolution of a system with respect to a usually global logical or physical clock, a concurrency relation defining the coincidence of time instants, and finally communication specifying the interplay of system entities. **Timed** models imply a total order of event tags, whereas in **untimed** models causality and data dependencies stipulate merely a partial order among them.

... time is introduced at mapping stage, as stated before.

In **synchronous** designs, all model elements run in lockstep and therefore state updates are carried out at the same time. This may be achieved with the abstraction of computation and communication taking zero time. Likewise one could think of the actual delay introduced by communication and computation as being irrelevant because they do not overlap. In **asynchronous** models, a new state is propagated directly between two individual elements and no global clock is available.	*. . . all components of the architecture model are synchronized to integer dividers of one global clock. The application model alone exhibits an asynchronous behavior.*
In the context of communication, the notion of synchronicity is used in another sense. Synchronous communication implies the simultaneous participation of entities in an action, which is blocking as long as at least one entity is not available. Asynchronous communication temporally decouples sending and receiving actions by involving a stateful mediator (for instance a FIFO channel storing the sent but not yet received events).	*. . . has native support for asynchronous communication exclusively, which however permits to emulate synchronous communication.*

This section confronted general properties of MoCs with the particular characteristics of DIPLODOCUS. To complement this survey, the following sections elaborate on discrete time MoCs most frequently referred to in related work of this thesis.

3.2.1 Finite state machines and Statecharts

Communicating finite state machines (FSM) constitute a widely used synchronous and timed model for the control flow of embedded systems. This representation explicitly specifies the system states and thus the amount of memory elements the system embraces. A traditional FSM used for engineering purposes is defined as a set of input signals, a set of output signals, a finite set of states, output functions relating output signals to the current state and the input signals, and a next state function mapping inputs and the current state to the subsequent one. FSM have their roots in the more abstract automata theory which is a branch of theoretical computer science. This theory does not discriminate input or output signals and rather emphasizes the acceptance or rejection of a sequence of symbols belonging to a formal language. Finite state machines however are more pragmatic in the sense that they have a direct representation in hardware: the two functions can easily be translated into boolean logic (combinational circuitry) and states are directly presented by memory elements (sequential circuitry).

Traditional FSM are not very capable when it comes to modeling concurrency and (data) memory: in that case, the so called state explosion problem is encountered. This latter problem could be mitigated by using several distinct FSM, which would correspond to

a single machine having a number of states equal to the product of the number of states of each machine. In addition to concurrency, hierarchy may be employed to cope with complexity. These two concepts have been introduced in the framework of Statecharts (synchronous, untimed) [44].
State machines constitute a good starting point for formal verification techniques such as model checking. The basic idea is to use algorithms that pass through the FSM with the objective to tag states which comply with a logic formula (CTL, LTL,..). Finally, a yes or no answer is obtained depending on whether the resulting set of states is empty or not. A large body of work exists on implicit representations, that yield a reduced memory consumption by circumventing the explicit enumeration of all system states.

3.2.2 Data Flow Networks

In **Data Flow Networks** (DFN), systems are structured by means of directed graphs comprising nodes interconnected by arcs. Nodes stand for concurrent processes performing operations on data and arcs symbolize event or data streams which are generated and consumed by these processes. DFN are often organized in a hierarchic way such that a process may be made of another graph. Synchronous data flow systems rely on the assumption that a constant amount of data is produced, consumed and processed per time unit. DFN are widely used in the context of signal processing and control applications (cf. Simulink).
Untimed, asynchronous **Kahn Process Networks** (KPN) [57] refer to a set of concurrent processes communicating through unbounded FIFO channels. As read requests are blocking, a process is suspended when reading from an empty FIFO and can be resumed as soon as data is written into the channel. Write requests are non-blocking by definition due to unlimited channels. Channels interconnect exactly one sending and one receiving process. The description of process behavior embraces sequential computations on local data as well as read and write operations. Unlike other MoCs, KPNs guarantee the useful property that resulting data streams solely depend on the input streams regardless of the order of process scheduling. However, this property comes at the expense of flexibility: a process is not able to verify the presence of input data to prevent a preemption. Furthermore, deadlock free execution is guaranteed as well by the MoC.
Petri nets [98] bear some resemblance with DFN and state machines. This MoC puts emphasis on the interaction of systems rather than on the functionality by describing systems in terms of places with transitions between them. To cause a transition to fire, a given number of tokens must be available in the input place. The state transition is finalized when tokens are generated in the output places. As firings of transitions may occur at the same time, the MoC naturally captures concurrency.

3.2.3 Discrete event

Even though the discrete event (DE) MoC (timed, asynchronous) [78] has no formal underpinning, it plays a very important role in hardware design and simulation in general. Simulation semantics of languages such as Verilog, VHDL and SystemC are defined with respect to this MoC. Events trigger processes at the time indicated by the time stamps of the former. In turn, processes may give rise to further events. A scheduler maintains a global event queue which is totally ordered with respect to the time stamps of the events. The scheduler elects the receiver process of the top-most event in the queue for execution. If time stamps are split into two parts, one standing for the physical part, one indicating the number of times a process has been evaluated, concurrency can be emulated on single processor systems. In VHDL, delta cycles and the evaluate-update policy for signals ensure the causality of concurrent processes.
Obviously, simulation engines based on DE perform very well when the modeled system is moderately active. An increasing activity leads to a considerable overhead due to event ordering and processing.
The DE MoC suffers from mainly two flaws: potential indeterminism introduced by simultaneous events and zero delay feed-back loops. However, the semantics of languages on top of the DE MoC such as VHDL may mask this indeterminism. DE models are still considered as the workhorse of simulation as many wide spread simulation environments are built upon them.

3.3 Classification

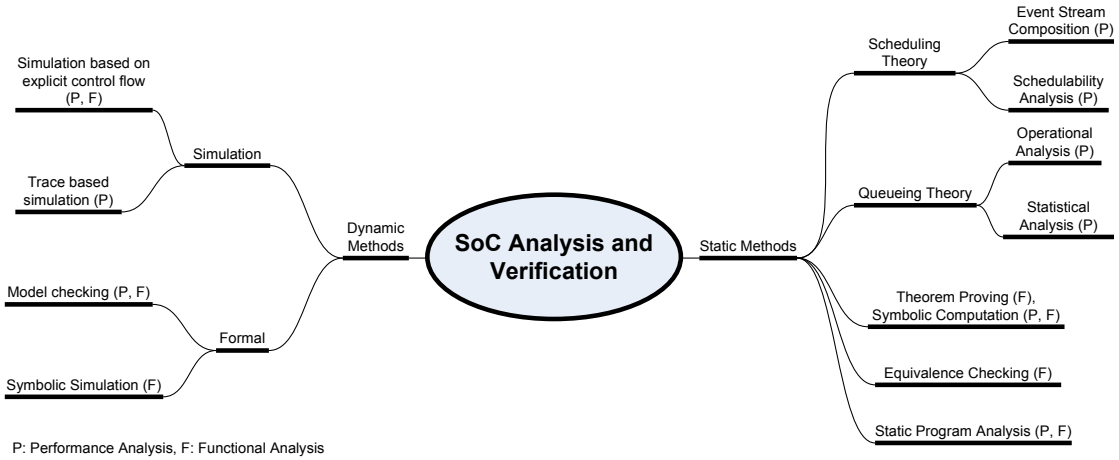


Figure 3.2: Classification of Verification approaches

Figure 3.2 constitutes an attempt to classify the huge body of work yielding verification approaches for SoC. Thereby, both approaches yielding functional verification (marked

with 'F') and performance evaluations (marked with 'P') are considered. The most evident and most discriminating factor is whether a model needs to be executed in order to produce results. The execution of a model is informally defined as the traversal of a (possibly infinite) sequence of states, constituting a subset of the space spanned by all possible valuations of the memory elements of the model. Therefore, the model must stipulate a series of state transitions/instructions to be carried out in a defined order, referred to as control flow.

Static methods usually do not define a sequence of transitions but embrace a list of entities instead (be it tasks, clients, servers, logic gates) which are characterized by a set of equations or key figures. (In the case of static program analysis, the input models yet contain a sequence of transitions, which however do not get executed according to the previous definition.)

Schedulability analysis defines a system as a set of tasks with properties such as best case execution time, worst case execution time, priorities, deadlines, etc. As composability is of paramount importance, the theory has been extended to cover the interplay of components providing and requiring service respectively.
In Figure 3.2 this technique is referred to as **Event Stream Composition**. The underlying assumption is that events do not have to be considered individually but may be gathered to streams governed by a certain regularity. Both schedulability analysis and Event Stream Composition yield performance figures and shed light on whether a system may cope with a given workload under certain constraints.

Equivalence checking is usually applied to digital electronic circuits at RTL level or even below. It is capable of (dis)proving that two representations of a circuit present the same behavior, from a black box perspective. Due to the sheer complexity of the problem, the topology of memory elements is constrained to be quite similar in both designs. This method is not discussed any further as this work is concerned with higher abstraction levels.

Queueing theory is the mathematical foundation for describing queues of entities awaiting service. The theory distinguishes several fundamental events such as arrival at the queue, waiting in the queue and being served. On the one hand, the outcomes of the theory consist of performance measures such as the average waiting time and the number of waiting/serviced entities. On the other hand, probabilities of particular system states (queue empty/full, encountering a specific waiting time) may be deduced.
Operational Analysis has evolved from queuing theory due to the observation that this theory even applies quite well if some of its assumptions are violated. The latter require a system to be studied in equilibrum, stationarity, and ergodicity and to present exponentially distributed service times. Operational analysis [34] relaxes the assumptions inasmuch as statistical values, necessitating a thorough analysis of many experiments, are replaced by parameters being directly measurable in practice.

Theorem Proving defines a system as a set of statements or axioms assumed to hold and therefore not requiring any proof. A theorem prover is able to determine whether some statement is a logical consequence of the axioms. The axioms could for instance describe valid actions of a system and the statement to be proved could define an undesired state. That way, the compliance of the system to constraints can be mathematically established.

As its name suggests, **Static Program Analysis** attempts to obtain information on a program without actually executing it. Static analysis may result in mathematical proves of properties ("Is the value of x twice the value of y after 3 iterations..."), hints to improve program performance (reformulation of statements, elimination of unused variables or dead code, data dependency information) or performance measures such as worst case execution time, loop bounds. Even if many problems are undecidable in general due to the halting problem, in practice programs often have nice properties allowing static analysis to succeed.

Symbolic computation is capable when it comes to studying systems for which mathematically defined MoC (Petri Nets, Kahn Process Networks, etc.) or terms (closed expressions, difference/differential equations) exist. For example, reachable markings in Petri Nets can be calculated with the aid of matrix multiplications. Symbolic computation is also heavily used at low abstraction levels, such as electronic circuits or analog systems

As stated previously, dynamic methods explicitly carry out system transitions stipulated in the model. In the presence of loop-like structures with variable conditions, the procedure does not necessarily terminate.

Model checking can be considered as a simulation technique that explores all reachable system states in a brute-force manner. Proofs of formulas in temporal logic are achieved by traversing the state space and recursively marking states that satisfy the property. Data-intensive systems should be abstracted before being checked, to avoid state space explosion. Thus, especially control flow centered systems are amenable to model checking techniques. Whenever execution comes to a point in the control flow providing several options to continue, all branches are traversed in a defined order. To limit runtime to a reasonable extent, branches are abandoned as soon as a recurring state is encountered. Therefore, an implicit or explicit representation of all traversed states has to be maintained. As a rule of thumb, an explicit representation entails the presence of a piece of data in a data structure for each encountered state. An implicit representation involves a boolean equation (often implemented in the form of a Binary Decision Diagram) evaluating to true if the binary state vector of an encountered state is plugged in.

Symbolic simulation covers several possible executions of a system at a single run. The

method is closely related to the notion of equivalence classes, denoting sets of elements which are equal with respect to an equivalence relation. For instance, given that program behavior depends on the equivalence relation $sgn(x)$, we do not have to simulate the program for each value in the range R of the datatype of x. Instead, we would conduct a symbolic simulation for the three equivalence classes of the quotient set of R, namely for $x > 0$, $x < 0$ and $x = 0$. Symbolic Simulation has the potential to greatly reduce the complexity of verification problems. However, special measures have to be taken to cope with loops exhibiting dynamic bounds.

Finally, the most popular and widely accepted verification technique in practice is **Simulation**. The lack of mathematical rigor comes with the advantage of making the technique easily accessible and applicable. The other side of the coin is that the absence of errors cannot be proved as the system is usually not analyzed exhaustively. While the principles of simulation are easy to grasp, reaching a sufficiently high level of confidence for results is an art. Therefore, various metrics mostly based on a certain type of coverage (be it statements, branches, functions, decisions, etc.) have been thought of. For large, partly critical systems, acceptable results from both performance and reliability perspective are obtained by combining simulation with formal techniques.

Subdividing the simulation of a system into several (often two) stages and cascading the corresponding simulators is referred to as **Trace based Simulation**. Thereby, simulation results of one simulator, the so-called traces, are considered as an input to the subsequent simulator. The subparts of the system are required to be independent, hence the traces provided by a simulator must not depend on decisions taken in the subsequent simulator. The system may be partitioned such that functionality and timing constraints introduced by an architecture (Y-Chart approach) are considered separately. That way a trace reflects the computation and communication demand (workload) encountered during a particular execution of the functional model. In that case, the method constrains the functional model to be time invariant. For instance, if the control flow depends on the number of samples stored in a communication channel, varying the timing may result in different traces. As the notion of time is not introduced until traces are generated, this leads to a contradiction making trace based simulation ill-suited. Obviously, trace based simulation proves its superiority as compared to conventional simulation if traces can be reused for many experiments.

Except for simulation, the presented verification techniques are mostly tied to a particular MoC, i.e. a set of rules governing the underlying system model. However, simulation is applicable at a broad range of abstraction levels characterized by their particular MoC.

3.4 Verification techniques

3.4.1 Formal and static methods

This section surveys related work based on dynamic, formal as well as on static techniques and covers all branches in Figure 3.2 except for simulation. In the following, pros and cons of formal and static methods are summarized:

+ exhaustive coverage of the design space captured by the system model
+ mathematical proofs of correctness are offered
+ possibility to spot corner cases for which suitable test vectors for simulation would probably not have been discovered
+ little effort (as compared to finding test vectors in simulation) is needed to set up formal verification with industrial tools

– The state explosion problem is encountered very quickly even for small-sized designs.
– Data abstraction is crucial to avoid state explosion: in hardware design, data is often abstracted to its presence or absence.
– A decision must be made concerning the main objective to be achieved: representing mainly the control flow of the system (data abstracted to the mere presence/absence vising equivalence checks or model checking) or focusing on the data flow (data abstracted to key figures such as throughput, arrival patterns,. . . vising traditional scheduling methods).
– Application models often have to be simplistic to be explored in a reasonable amount of time and with a limited amount of memory.
– Only few engineers have expertise in handling formal verification tools. As the latter have also had the reputation of belonging to the verification "nerd" domain, engineers are reluctant to learning formal techniques.
– Final executable models (e.g. source code) are normally not amenable to formal verification techniques any more; there are two viable solutions: automatic code generation being correct by construction from higher levels of abstraction, or formal verification of subparts.

3.4.1.1 Event Stream Composition

The so-called SymTA/S approach [42; 48] relies on classical methods for real-time scheduling analysis to obtain performance measures of distributed systems. The acronym stands for "Symbolic Timing Analysis for Systems". The behavior of the environment is modeled by means of standard event arrival patterns including periodic and sporadic events with jitters or bursts. The main contribution is the extension of the scope of scheduling theory for mono-processors. Event streams are propagated among resources of distributed systems in a way that each resource may be analyzed separately with classical algorithms. However, the applicability of scheduling theories requires the task model to be simplistic and thus it merely reflects best case and worst case execution times. Control flow within tasks cannot be considered at all.
[28] Real Time Calculus has evolved from Network Calculus [1] and aims at determining the performance of distributed systems by analyzing event streams connecting resources. Any deterministic event stream can be modeled with the aid of arrival curves denoting lower and upper bounds for event occurrences. As compared to SymTA/S, this approach is not limited to standard event patterns which turn out to be a special case of this framework. Service curves are used to model hardware by accounting for the availability of computation and communication resources. Concerning detailedness of the task model however, it suffers from the same drawback as SymTA/S. It may be tedious if not impossible to model tasks exhibiting data dependent or irregular behavior.

3.4.1.2 Operational Analysis

As mentioned earlier, operational analysis [34] has its roots in the mathematically well founded queuing theory. [113] establishes performance laws that relate processor speed changes in parallel computing systems to variations of job queuing time. As opposed to queuing theory, only measurable, so called blackbox-observable information is leveraged to predict the behavior of a processor. [59] looks into the performance of parallel servers at application level and allows for modeling multi-processor systems. The authors come up with laws predicting the impact on performance of a varying number of servers. Operational analysis is very capable when it comes to examining coarse grained system performance based on weak assumptions which are easy to satisfy. System behavior should follow regular patterns, that is the number of job arrivals occurring during an observation period and the service time of a task should not be subject to considerable fluctuations. As for any static technique, control flow and data dependencies cannot be accounted for.

3.4.1.3 Symbolic Computation

[81] presents a framework for computation and communication refinement for multi-processor Soc Design. Stochastic automata networks model the application behavior

[1] a profile based on Network Calculus is supported by TTool [15]

and the authors claim that this formalism enables fast analytical performance evaluations. Analytical techniques spare the developer time consuming profiling simulations for predicting power and performance figures. When it comes to mapping an application onto an architecture, transitions and states have to be added to the application model. Therefore, application and architecture matters are not strictly handled in an orthogonal fashion. Due to a lack of data abstraction, the modeling of memory elements can quickly lead to the state space explosion problem.

3.4.1.4 Static program analysis

A program slice is defined as a subset of statements of a program that may have an impact on a particular statement or value. The program is thus sliced with respect to this slicing criterion.
[108] makes use of this method to infer upper bounds of program execution, so called wort case execution times (WCET). WCET are crucial for dimensioning embedded systems and may also open the door to other static techniques. In this context, program slicing reduces the significant part of a program by revealing statements and variables which demonstrably do not have an influence on program execution. The presented effort is a good representative for a huge body of work concerned with the static analysis of program behavior for performance studies.
[125] provides means for formal and simulation based evaluation of UML/SysML models for performance analysis of SoCs. UML Sequence diagrams constitute the starting point for the functional description. They are subsequently transformed into so called communication dependency graphs (CDGs) which capture the control flow, synchronization dependencies and timing information. CDGs are in turn amenable to static analysis in order to determine key performance parameters such as best case execution time (BCET), WCET and I/O data rates. A drawback of this approach is that data flow independence has to be kept, thus preventing case distinctions and loops with variable bounds to be part of the application model.
[76] is also dedicated to determining bounds on program runtime (WCET) on a given processor. As opposed to [108], another formalism is utilized which bears resemblance with the Kirchhoff's nodal rule. Constraints are established for basic blocks relating the control flows entering it and the control flows leaving it respectively. Together with constraints provided by the user to denote loop bounds a solver calculates the maximum run time of the entire program.

3.4.1.5 Symbolic Simulation

Formalization methods suggested in [66] pave the way for synthesis, performance analysis and verification based on high level models. To that end, C++ code is transformed into data flow graphs (DFG) which store a symbolic representation of variables. During the symbolic execution of the program, each variable assignment triggers an update of the corresponding symbolic expression. As no assumption of the concrete value of

variables is made, conditional branches imply that both paths have to be taken. Each explored branch is assigned a symbolic condition. Limitations of the approach are encountered as soon as loops with variable bounds come into play. Implicit loop unrolling can lead to size explosion of the DFG holding the symbolic values of variables.
[31] describes a similar application of Symbolic Simulation which is more tailored to low level models, that is assembly code of embedded software. The authors primarily target verification and more specifically equivalence checks of different versions of a program. To reduce the inherent complexity of the task, function calls may not be interpreted, i.e. the function is not executed but treated symbolically as a mathematical expression. In analogy with the previous work, the verification of large programs is hampered by the sheer complexity of the problem.
[79] introduces a formal executable system model based on communicating tasks interconnected with FIFO channels. Tasks are considered to be atomic and they are characterized by the time interval [BCET, WCET]. The authors claim to bridge the gap between simulation and formal verification by performing discrete event simulations of the system model. Execution traces are provided by means of an event order tree. The latter captures the indeterminism of the MoC and thus represents a symbolic representation of all distinguishable execution traces. Each path in the event order tree stands for different constraints on the execution intervals of tasks.

3.4.1.6 Model Checking

[47] relies on timed automata to analyze timeliness properties of embedded systems. The UPPAAL model checker is used to evaluate the automata which must be created manually. There is no automated translation routine from a high level language (UML,...) and thus the creation of the automata turns out to be error prone.
As shown in [126], not only formal models, but also software programs may be the starting point for model checking. The authors believe that the focus of the model checking community on formal languages should be shifted towards main stream programming languages. Despite formal verification of high level models and correct by construction methodologies, the final code is often modified by hand. Bugs are hence introduced at levels formal tools do not deal with. The paper proposes solutions to some intricate problems of model checking (JAVA) programs: mastering the huge state space spanned by all used variables, extracting state vectors from the virtual machine and collapsing them, leveraging data abstraction (predicate abstraction) and static analysis for pruning the state space. The work is built on a custom model checker steamlined for JAVA input models.
Another way of doing is to perform language translation and to rely on existing model checkers. [50] attaches a formal semantics to SystemC models and transforms them into UPPAAL timed automata. The methodology preserves the behavioral semantics and the structure of a SystemC design while the model transformation is accomplished automatically.
[51] extends the scope of the previous attempt by reconciling Model Checking and Sys-

temC simulation. It provides means for conformance testing for SystemC models at different levels of abstraction. High Level SystemC models may be automatically translated into timed automata supported by the UPPAAL model checker. Traces obtained from that formal model are compared against traces obtained from refined SystemC models in order to determine whether properties proved for the high level model still hold after refinement. However, as compared to SystemC, DIPLODOCUS pushes abstractions further by enforcing the separation of application and architecture and thus providing means for efficient Design Space Exploration.

3.4.2 MoC-centric methods

Some environments place emphasis on the interplay of several MoC. The verification methods does not play a major role, attention is rather turned to bridging the semantic gap between MoCs. For this reason, a special section is dedicated to the following approaches:
The PUMA [132] framework is a unified approach to software modeling. It provides an interface between high level input models (such as UML diagrams) and performance oriented models. For that purpose, input models are first translated into an intermediate format called CSM so as to filter out irrelevant information for performance evaluations. In a second step, CSM can be converted to Petri Nets, Markov models, etc., and the resulting performance figures and design advice is fed back to the initial model. However, this framework concentrates on the modeling of software and thus does not yield a mapping where functionality is associated to software or hardware elements.
Ptolemy [36] is a design framework targeting modeling, simulation and design of embedded systems with emphasis on the integration of different MoC. Being a very general framework, Ptolemy is able to deal with all kinds of systems and is not especially tailored to the field of SoCs. So called domains encapsulate different MoC and comprise concurrent functional units named actors. A domain is also equipped with a director which is in charge of the control flow management, meaning that the order of execution of actors. Data flow is implemented using receivers which define the communication semantics between actors. Interface automata contained in receivers and directors bridge the semantic gap between between different MoCs. As the modeling methodology of Ptolemy mainly captures functionality, it lacks capabilities for performance modeling such as resource sharing and contention.
BIP [22] describes a MoC which is aligned along three main axes: **B**ehavior expressed in terms of transitions, **I**nteractions between transitions and **P**riorities arbitrating mutual exclusive interactions (in analogy to a scheduling policy). BIP places emphasis on the assembly of models in terms of components while preserving properties so as to facilitate analysis and transformations across heterogeneous model boundaries (timed/untimed, synchronous/asynchronous, etc.). Due to the separation of concerns, system construction can be considered as a superposition of elementary transformations along one of the three axes. As opposed to other environments, the axes are really independent

which makes that any point in the three dimensional design space is meaningful. BIP models are both amenable to simulation and model checking through transformation to C++ code. A combination of an execution engine and the generated code can be run in simulation or in state space exploration mode, the latter yielding graphs to be analyzed with model checking tools.

3.4.3 Simulation centric methods

3.4.3.1 Abstraction Levels

As stated earlier, simulation is an important and widely used technique for the verification of functional correctness of SoCs. Traditionally, simulation is classified according to the abstraction level of the underlying system model. Abstraction is defined as "the cognitive process of isolating, or abstracting, a common feature or relationship observed in a number of things, or the product of such a process." [1]. In the domain of electronic systems, the designer has to isolate details of the system that are relevant to the properties to be (dis)proved. Even if properties refer to system specific entities (like tasks, signals), they are often quite similar in nature (compliance to deadlines, temporal order of actions to be respected, etc.). This insight has stimulated the emergence of several commonly agreed levels of abstraction, trading-off differently accuracy and simulation speed. Given the variety of circulating definitions, the attempt to precisely define abstraction levels is doomed to failure.[2] The following overview represents the common denominator of definitions found in the related work of this thesis.
The chosen abstraction level presupposes some properties of the MoC to be applied. The expressiveness of languages may be restricted to a single MoC (such as the "Petri-Net-Language" to Petri Nets) or range from purely functional description to the cycle-accurate level (such as SystemC).
Furthermore, simulation speed is an issue of great concern and depends on the simulation engine as well as on the detailedness of the input model. Hence, moving down in the abstraction hierarchy means refining a model to the detriment of simulation time. Several levels of abstraction are usually considered in the context of embedded system design:

System Level design is defined as "the utilization of appropriate abstractions in order to increase comprehension about a system, and to enhance the probability of a successful implementation of functionality in a cost-effective manner using generic architecture and abstract application models" [20]. Of course, this concept is destined on the one hand for early stages in the design flow where the specification still lacks many details. On the other hand, detailed models could be abstracted with the aim of extending the scope of simulation or increasing simulation performance. Two complementary

[1] Online issue of Encyclopædia Britannica

[2] This is however not the case for Models Of Computation (see Section 3.2) which have successfully been integrated into a formal framework [74; 82].

kinds of models have to be discriminated: purely functional models aim at assessing (mathematical) algorithms and the application logic. High level and sometimes domain specific languages such as Matlab, Simulink, C++ have proven their applicability for that purpose. These models do not comprise any reference to the target platform, albeit of course being constrained by the host platform used for their execution.
Other frameworks rather emphasize control or data flow and HW/SW partitioning: applications are abstracted such that they are not functional any more. Those models may be tailored towards the data or control flow depending on the respective MoC. Functional entities can be associated to rudimentary and generic hardware components so as to obtain first performance figures. The aforementioned functional reference models can serve as a good starting point for the estimation of algorithmic complexity. Qualified annotations enhance accuracy of the non-functional performance models.

Transaction Level Modeling (TLM) highlights the concept of separating communication from computation within a system [41]. Modules, represented as communicating concurrent processes, exchange information by means of transactions rather than driving signals. As signal handling and update mechanisms are very time consuming in simulation, TLM also yields performance improvements. Transactions are conveyed on channels providing standardized interfaces, which facilitates the encapsulation of communication protocols in separate modules. Even if concrete communication protocols are incorporated, emphasis is placed on the functionality of data transfers. Thanks to the standardized interfaces, communication protocol implementations are interchangeable and independent from the application. TLM is applied with different semantics with respect to the granularity of data in communications and timing accuracy (timed, untimed, intermediate variants).

Instruction Set Level models are applied in the context of programmable computing devices. The underlying idea is that intermediate states and transitions of a device while performing computations may not be of interest for the designer. This level solely captures the system state (register values, flags,...) at the end of computations. The latter are defined by a so called instruction together with corresponding operands and addressing modes. Simulators are usually called instruction set simulators and involve two machines: the host processor on the one hand and the target processor on the other hand. The host processor executes the Instruction Set Simulator, whereas the simulator constitutes a model of the target processor.

The **Register Transfer Level** is a widespread abstraction level focusing on the operation of digital circuits. The latter are described in terms of storage elements, called registers and combinational functions relating the current and the future state of registers. Data is propagated by means of signals among hardware entities. Hardware description languages such as Verilog and VHDL as well as SystemC are well suited for register transfer modeling. If the RTL model complies to certain rules, Verilog and VHDL also target an automatic translation into a netlist (gate level description).

Gate Level simulators take netlists consisting of interconnected logic gates (NAND, NOR, etc) as an input and perform functional simulation as well as timing analysis. Much effort has been spent on improving the performance of the first logic simulators.

The **Switch Level** abstracts transistors as on/off switches which may be used to obtain more accurate delay estimations for logic gates.

The **Transistor Level** is the most accurate and complex representation for electronic circuits. Simulations (for instance using the SPICE simulator) are very time consuming as the nonlinear dependency between voltage and current within transistors is accounted for by mathematical models. Current and voltage may be determined anywhere in the circuit using algorithms similar to nodal and mesh analysis.

A brief overview of pros and cons of simulation based techniques is given below:

+ scale very well with design size as opposed to formal methods
+ may deal with executable, realistic models that capture system performance very accurately
+ the control flow within models can be very detailed and complex, real data samples are taken into account
+ represent the tried and tested workhorse of verification, a lot of engineers are familiar with simulation techniques which are widely adopted and accepted
− big effort needed to create test benches, to identify test vectors and to determine the coverage
− do not scale at all with coverage, the incremental effort required to achieve each percentage point of coverage beyond the first 90 percent increases exponentially with simulation
− do not provide the measure of confidence obtained with formal tools and their mathematical proofs

3.4.3.2 Explicit Control Flow based methods

UML based methods:
[54] elaborates on the integration of UML-based behavioral patterns into executable functional and simulatable models. The authors developed an extension to Ptolemy in the form of a domain which achieves the integration of the UML syntax and semantics. However, as simulation in Ptolemy are purely functional, it is not obvious how to apply the methodology in the context of SoCs. Separation of application and architecture

matters cannot be achieved easily.
[124] presents a UML modeling framework based on an existing UML modeler which allows for designing real time and embedded systems. The lowest abstraction level among the three supported ones enables code generation. It seems that simulation and performance analysis cannot be carried out at the highest abstraction layer, as it is the case for our modeling environment. No details are given concerning the underlying simulation strategy.
[70] elaborates on a UML2 based environment for platform modeling and performance evaluation. Workload models can be developed in UML in a hierarchical fashion or in SystemC, whereas platform models must be described in SystemC only. Behavioral descriptions of application entities have to be provided in the form of state machines. They are abstract in the sense that they do not perform computations of the real application. Simulations rely on the SystemC workload model generated from UML and the SystemC component models taken from a library. The latter are timed in a cycle approximate fashion.
[110] introduces a virtual machine which is amenable to the execution of a UML subset (Class, State Machine and Sequence Diagrams) for embedded software and reconfigurable hardware. Hence, emphasis is put on executable models without making the distinction between application and architecture issues. UML specifications are compiled to equivalent binary representations based on an instruction set supporting object management such as instantiation, destruction, operation invocation. The virtual machine which executes the models can be implemented on software or hardware platforms.

Lower abstraction levels:
The work discussed in [105] bases on the Y-Chart approach. Applications are modeled as concurrent tasks which have to be specified in C++. The architectural composition of the system and the mapping of tasks on processing elements is provided in a separate file. A library contains predefined hardware elements written in SystemC (TLM level) which constitute the building blocks for an architecture model. The usage of graphical high level languages such as UML is not supported so that the development of C++ models of application functionality may turn out to be cumbersome.
[10] addresses Design Space Exploration at ASIP level and focuses more specifically on architecture exploration. The LisaTek design platform constitutes an abstraction layer above the RTL level usually expressed in VHDL or Verilog. The approach is centered around processor models written in the architecture description language LISA. The framework permits the automatic generation of development tools such as instruction-set simulator, C-compiler, assembler, and linker. LISA models may also be transformed into VHDL, SystemC and Verilog representations as well.
[130] puts emphasis on the gradual and simultaneous refinement of processor models written in LISA interconnected with bus models expressed in SystemC. Abstraction levels range from transaction level to RTL level. Our framework however is settled at a higher level of abstraction and primarily targets design space exploration at system level.

Data flow oriented methods:
SystemCoDesigner [45] supports automatic design space exploration and creation of hardware accelerators. The application domain is restricted to multimedia and networking due to the applied MOC. The latter relies on finite state machines specifying the communication behavior and controlling methods. Data consumption and production may only take place after the execution of the methods. The SystemC model can be transformed into hardware accelerators and software modules. Hardware modules are generated by means of the Forte Cynthesizer. Graphical high level languages such as UML are not available for system modeling. Furthermore, application and architecture are merged in one model.
[17] bears resemblance with our approach with respect to the modeling language UML which is applied for architecture, application and mapping models. As opposed to our framework, the focus is put on streaming applications that make scarce use of control messaging and branching. For this reason, the semantics of Kahn process networks has been adopted for application models. Simulation is carried out on SystemC TLM level, thus it does not leverage all abstraction applied at the modeling stage.

Non-UML system level methods:
[107] also follows the Y-Chart approach. Mapping of applications onto architectures is performed automatically on graph-based descriptions and relies on side-information provided by the system designer. Application models are not based on functional descriptions but only on abstract information (such as cycle counts). They are therefore independent of the modeling language. The aforementioned framework is not intended for formal analysis.
The Metropolis [21] metamodel language enables the use of different kinds of MoC. Designs may be described at different levels of abstraction and subsequently be simulated or verified. Processes communicating through media constitute the application model. Constraints can be used to restrict possible executions in case of non-deterministic behavior. Architecture is accounted for by performance models associating a cost to events. A mapping network establishes the link between events generated by the application model and the underlying architecture model.
Koski [58] permits automated design space exploration and FPGA synthesis based on orthogonal application and architecture modeling at system level. The input specification is given as a Kahn process network modeled in UML, which may be refined with Statecharts. The environment focuses on code generation for the target platform based on libraries and does neither offer a trade-off between formal techniques and simulation nor means to graphically and formally express properties to be verified during simulation.
System level simulators such as VAST [4] and Coware [2] are typically platform based, they rely on a bottom-up modeling methodology and require specific models of architecture components (processors, buses,...) to perform transaction level simulation. In that context, Instruction Level Simulators (ISS) may constitute an adequate means to provide cycle accurate behavior of CPU models. Cofluent [1], [27] yet goes a step

further by introducing generic HW components being parameterizable by the designer and a partially graphical way to describe application functionality. The latter approach share common points with our architecture model because it is also generic and architecture exploration is accomplished on application models. But in addition to that, the DIPLODOCUS methodology is based on data and functional abstractions which may further reduce simulation time. Due to that functional abstraction, all involved models are completely graphical and no line of code has to be produced. Even verification can be seamlessly accomplished in UML by means of the TEPE language.
[69] presents a method for creating abstract instruction workload models from source code. The source code is abstracted by introducing the symbolic instructions read, write and execute. These symbolic programs can be used to speed up performance evaluations of systems modeled in terms of processing units, communication units and storage units.

3.4.3.3 Trace based approaches

SESAME ([77; 100; 101; 102]) stems from the earlier SPADE project and is developed in the context of the ARTEMIS joint undertaking. SESAME provides high level modeling and simulation methods for system level performance evaluation of embedded systems. As the methodology bases on the Y-Chart approach, concerns of architecture and application are separated. The applications are modeled in terms of Kahn process networks (KPN). Therefore, they are not able to reflect indeterminism and time dependent behavior. Traces produced by the application model represent the load for the architecture components. Architecture models operate at transaction level and are implemented in SystemC or Pearl. Generic building blocks for architecture components are gathered in a library. The designer may be guided towards an optimal architecture by means of multiobjective optimization techniques.
[117] attempts to compensate the general weakness of trace driven simulation to evaluate systems under a single application workload. It thus acknowledges the fact that today's embedded systems are operated in different use cases and therefore also the applications' resource demands change drastically over time. The paper advocates the usage of hierarchical scenario databases: the multi-application level defines the configuration of independent active KPN, the KPN level describes which combination of traces of the Kahn processes are valid execution traces of the KPN, and the lowermost level captures the respective traces which may also contain statically bounded loops. Moreover, the paper elaborates on optimization techniques to detect iterations in traces to reduce the total memory space of the data base.
[131] describes a trace based simulation environment based on SystemC hardware models. The underlying principle is to abstract the involved functionalities by their processing latencies so that the corresponding program code does not have to be run at all. Accurate hardware models interpret the traces and forward packet references through the system. In this framework, source code of all applications must be available to obtain initial traces of tasks. The control flow of involved applications does not have

to be time dependent, otherwise initial traces could not be determined independently from the architecture.
[114] presents a SoC modeling methodology which does not offer a clear separation of architecture/application. Software tasks are described with the aid of a textual task description language whose operators bear resemblance at first glance with those available in DIPLODOCUS. Symbolic instructions account for communications (read/write operations) and computations. However, commands directly reference hardware components (for instance the memory data is read from) and hence the Y-Chart approach is violated. The trace driven simulation relies on optimized SystemC models at TLM level. The whole modeling procedure is accomplished in textual form - no graphical interface has been defined.

3.4.4 Hybrid Static/Simulation methods

Generally, an issue of great concern of informal verification techniques is to increase design space coverage so as to evaluate more design points. Combining formal techniques with simulation always results in an incomplete technique due to the lack of exhaustiveness of informal techniques. In the following, frameworks are introduced which employ analytic techniques with the objective to speed up simulation, to enhance its coverage and to avoid the state explosion problem.
[26] speeds up the conventional simulation approach for interactions between shared resources. The work advocates a hybrid simulation kernel which applies simulation to capture system dynamics together with an analytical approach. The analysis bases on C programs enhanced with annotations indicating the computational complexity of sections. The basic simulation concept is centered around the principle that contention of shared resources is ignored between software annotations. Accesses to shared resources within one annotation region are grouped and managed by an analytical model which assigns time penalties to each competing logical thread. The penalties are taken into account during future calculations of the physical time. The approach concentrates on task scheduling on processor-based systems and does not model HW accelerators and the impact of communication topologies.
[71] suggests a hybrid approach which paves the way for combining analytic and simulation techniques. Event streams are defined by upper and lower arrival curves according to the notation used in Real-Time Calculus. Interfaces are defined which permit the conversion from event streams from the simulation subsystem into an event model for formal analysis and vice versa. Hence task chains may be broken down into segments which are treated separately on potentially different hardware resources and with different analysis procedures.
[96] sheds light on the combination of event stream analysis and dynamic, hence state based models. The approach acknowledges the fact that static analytical approaches tend to be overly pessimistic by assuming a stationary behavior of system components.

As system states cannot be expressed by definition, variability in the workload has to be captured by average, pessimistic WCET values. To alleviate this shortcoming, two remedies are proposed: considering workload variability automata when constructing arrival curves that represent the resource demand, and solving the problem in the stateful domain by transforming arrival curves to event generators expressed Timed Automata, performing model checking, and finally leveraging the results to derive output arrival curves. The methodology seems to be quite complex and thus its acceptance in practice depends on the availability of adequate tooling.

3.4.5 Communication centric methods

Communication centric approaches such as [61; 72; 73; 94; 133] are concerned with the analysis of SoC interconnect architectures (Buses, Crossbars, Networks-on-Chip, etc) and their impact on system performance. As opposed to DIPLODOCUS, computation architectures and the respective allocation of application tasks are ignored. Several frameworks permit an automatic exploration of a clearly defined design space, for instance communication channel mapping on a given bus topology.

3.4.6 Improving simulation speed

When it comes to improving simulation speed, two conceptually different ideas are prevalent. The first one affects the modeling methodology by subdividing the model into independent parts or augmenting its level of abstraction. In that case, simulation results may suffer from a loss of accuracy:

- **Trace based simulation**: models are decomposed into independent parts (cf. Section 3.4.3.3), which are simulated in a cascaded fashion. Simulation results from one stage, referred to as traces that are fed to the subsequent simulator which considers them as the workload. A performance gain is achieved by reusing traces for several subsequent simulations, for example providing the same workload to different architectures.

- **Model abstraction**: instead of a detailed simulation of the workload, the latter can be abstracted by means of symbolic instructions. Moreover, timing behavior may be abstracted by allowing local clocks of components to be in advance with respect to the global simulation time. This may or may not require concessions concerning accuracy, depending on whether possibly missed synchronization points are compensated.

- **Symbiosis of simulation and static techniques**: may have an impact on accuracy depending on the formal techniques used, see Section 3.4.4

The second idea is about increasing simulation performance while maintaining accuracy, at least to a large extent.

- **Simulation techniques**: technical improvements may target the simulation engine (e.g. the SystemC scheduler) or compilers (exploitation of multiprocessor platforms, handling of compiler inlining, etc.) used to build and simulate executable models.

- **Towards native execution**: in case source code is available, performance improvements may target the partial or full native execution on the host (simulation) machine. Parts of the model whose performance measures are not of interest are executed natively, while parts to be assessed are passed to an ISS. This raises the issue of capturing the system state on one side and restablishing it on the other. Another solution consists in running an annotated version of the application natively, which has been enriched with timing information of the target platform. However, it is an intricate task to establish the relationship between the assembly code of the application compiled for the target platform and the source code.

3.4.6.1 Timing abstractions

[116] points out an approach to temporarily decouple the clocks of SystemC processes so as to minimize the synchronization overhead with the Kernel and to increase simulation performance. The already existing SystemC QuantumKeeper merely allows a process to postpone the synchronization with the global simulation clock as long as the local time offset does not exceed a defined quantum. Issues concerning the dependency of temporally decoupled processes are completely left to the user. The authors suggest an extension called QuantumGiver, which keeps track of completed transactions since the last synchronization point. Conflicts are automatically detected and resolved transparent to the involved processes. Moreover, a hybrid dynamic/static method is described to augment the application with information about timing. If source code lines and binary level basic blocks cannot be statically related in an unambiguous manner, dynamic information about the predecessor of a basic block is taken into account.

3.4.6.2 Simulation techniques

[123] proposes a methodology to speed up simulation of multi-processor SoCs at TLM level with additional timing information. The idea of time propagation through transaction passing bears some resemblance with our new simulation approach. However, our environment is settled at a higher abstraction level so that the procedure of transaction passing has been extended to optimally support high level models. As our aim is very fast simulation from our specific UML models, our simulation engine is not based on the SystemC kernel any more.
In the scope of [97], a new SystemC kernel was developed to maximize performance and to avoid all unnecessary code. The kernel supports a subset of the standard SystemC language. In addition to that, the compiler inlining algorithm was overwritten to reduce the impact of virtual method calls and the handling of signal buffers was

optimized. A novel scheduling strategy addresses the problem of unnecessary wake-up calls of processes. Signal dependency information provided by the user is fed to a hybrid scheduling engine. The latter is intermediate between static and dynamic so that performance improves with the amount of available information.
The method presented in [87] aims at reducing the number of unnecessary output evaluations and wake-up calls of SystemC processes. Signal dependencies are automatically extracted from the SystemC source code. SystemC processes are subsequently split with the aid of this dependency information relating output signals and input signals of the process. The objective is to obtain processes containing only code that evaluates their own outputs. The SystemC scheduler is thus enabled to figure out the optimal order of process execution which prevents unnecessary output evaluations. Furthermore, the authors tend to limit the number of unnecessary wake up calls of processes by enhancing the method outlined in [97].

3.4.6.3 Towards native execution

[112] describes a hybrid approach consolidating analytic and simulation methods. Static execution time analysis is applied to get first estimates of the number of processor cycles consumed by basic blocks of the application. In a second step, dynamic effects are accounted for by a SystemC simulation of an annotated version of the source code. These annotations comprise the calculation of cache and branching penalties which are added to the statically determined cycle count at run time.
[68] aims at mitigating the issue of low simulation speed of ISS by means of a hybrid simulation framework. The latter allows switching between native code execution and ISS based simulation. Parts of the application are represented in the form of a coprocessor which is controlled by the ISS. The outsourced code is executed natively on the host machine so as to speed up simulation. The application source code has to be preprocessed for the native execution in order to assure a consistent handling of global variables, floating point operations, call of C library functions, etc.
An application transparent emulation of OS primitives is shown in [23]. Applications invoking OS routines may be executed by an ISS without having to simulate the whole OS functionality. Low level system calls are intercepted by the ISS and redirected to the host environment. An extension of this mechanism permits the emulation of concurrency management. This way, thread creation, destruction, mutex operations, etc can be carried out and controlled on the host system.
[128] tackles the problem of mapping binary code on source code to enrich the latter with timing annotations. In the presence of compiler optimization techniques, control flow at binary level significantly differs from control flow at source level. The authors rely on a three step approach consisting of the generation of mapping information, the generation of timing information and the instrumentation of the source code. The presented mapping algorithm matches the loop levels of binary and source code and is that way more accurate than approaches relying on code mapping.
[127] elaborates on speeding up HW/SW co-simulation by means of compiled simu-

lation. Standard ISS rely on an interpretative simulation technique which normally embraces a main loop. In this loop instructions are fetched, decoded and executed using a big case distinction statement. The compiled simulation approach directly translates target instructions into host instructions thus eliminating fetch and decode steps as well as interpretation overheads. Communication between a simulated processor and hardware elements is possible.

3.5 Property specification

One contribution of this thesis is the seamless integration of formal properties into UML based environments by means of the TEPE language. TEPE is appropriate for a wide range of system models originating from labeled transition systems. A labeled transition system consists of states and **transitions** between these states. Transitions are triggered by so called **events** which may either refer to a single atomic action or to a set of actions carrying the same label. In this work, the latter meaning of *event* is used synonym with the notion of **signal**, also subsuming a set of similar events.
This section reviews popular languages in the field of property verification, some of which are built upon UML while others define their own syntax. Especially in the hardware community, verification statements referred to as assertions are interwoven with the Register Transfer Level (RTL) source code and are closely tied to clock cycles as a sampling event. That means, whenever it comes to defining the temporal scope of conditions, the notion of clock cycles is taken as a reference. TEPE puts emphasis on the temporal and logical relation of signals and properties, without attaching an outstanding importance to a particular signal.

3.5.1 Non-UML approaches

System Verilog [5] provides concurrent assertions for describing behavior that spans over time. The underlying event model is based on clock ticks. TEPE constraints however operate on physical or logical time for property verification.
The e-language [122] somewhat extends the System Verilog event model by introducing user defined events derived from behavior or other events. However, temporal expressions require a trigger event to be selected for condition evaluation. In TEPE, constraints may specify several sampling events (e.g. signals) which may evolve over time.
PSL [6] can be considered as an extension of LTL and CTL temporal logic and the expressiveness of its temporal layer resembles the System Verilog specification language. The boolean layer of PSL embraces a default clock declaration. This declaration is implicitly referred to in all properties and temporal expressions that are not annotated with a clock. So called "properties" are used to describe behavior over time and they are made up of a Boolean expression and a clock expression amongst others. However, the

aforementioned languages fail to model physical time independently of clock cycles.
The SystemC Verification Standard [90] addresses the creation of test benches and allows both for random stimulus generation and recording of resulting transactions. To our knowledge, it does neither comprise a syntax for expressing temporal properties, nor automated ways to verify them.
[115] advocates a nice graphical notation which aims to simplify the formalization of requirements for model checking. System executions are expressed in the form of time-line diagrams discriminating optional, mandatory, fail events and related constraints. As for other trace-based approaches, conditional or varying system behavior cannot easily be expressed. Moreover, the approach does not address real-time or performance requirements.

3.5.2 UML approaches

The MARTE profile embraces the Value Specification Language VSL [92]. The language alone is not able to describe valid system executions: its goal is to verify the values of constraints, properties and stereotype attributes particularly related to non-functional aspects. When used in combination with sequence diagrams, VSL does not compensate the poor formal expressiveness of the latter. Live Sequence Charts [33] [43] address this issue by discriminating mandatory from provisional behavior. However, capturing a set of acceptable traces still relies on condition and loop primitives of conventional sequence diagrams and is therefore cumbersome. Additionally, the integration of equations that have to be fulfilled as a function of the system behavior is not straightforward in UML and requires the usage of OCL, thereby circumventing the graphical notation.

The MARTE Time model has defined an unambiguous time structure which can be leveraged to build precise timed models amenable to formal analysis. A corresponding concrete syntax is the clock constraint specification language (CCSL) [80] [9], which describes system events of different types as abstract clocks. The language allows to reason about relationships between these clocks, for instance periodicity, precedence alternation, etc. The work inspired ours as it provides a solid theoretical foundation for sequential and time constraint modeling. However, TEPE does not require the user to abstract events to clocks. The user just has to identify structures similar to signals and attributes. This procedure should be straightforward for formalisms stemming from labeled transition systems. To specify time constraints, TEPE does not demand for a clock whose ticks are interpreted as time advancements. The tool suite supporting TEPE focuses on the verification of behavior defined in other formalisms (such as Activity Diagrams or State Machines). As opposed to that, CCSL tooling targets observer generation in VHDL and Esterel as well as animation of UML models where CCSL formulas serve as behavioral descriptions.
Moreover, our objective was to suggest a concrete syntax (based on UML/SysML) paving the way for a seamless integration into various modeling environments.

3.5.3 Tooling

The Rhapsody tool used by [32] similarly enables formal verification of SysML diagrams using UPPAAL. Unlike TTool, Rhapsody does neither distinguish between requirements and properties nor does it support a property expression language - such as TEPE - and computation operators in state machines. In terms of user-friendliness, TTool allows one to right-click on an action symbol and automatically verify the reachability of that action. In the same situation, the user of Rhapsody is obliged to enter a logic formula, which assumes some knowledge in logic. The OMEGA2 environment [89] has also strong connections with Rhapsody for it implements the same semantics. OMEGA2 supports requirement diagrams as defined in SysML. Conversely ARTISAN [46] extends SysML to cope with continuous flows. ARTISAN models may contain probabilities and interruptible regions, two concepts not yet supported by profiles compatible with TTool. Electronic System Level (ESL), which is an emerging electronic design methodology, has stimulated research work on joint use of SysML and formal languages supported by simulation tools. Several papers discuss solutions where a model is designed in SysML and translated into VHDL-AMS [3] or Simulink [121]. Mechanical engineering is another area where SysML is combined with already existing domain specific languages, such as Modelica or bond graphs.
TTool supports several UML profiles, in particular TURTLE [13], DIPLODOCUS [14] and AVATAR [65] and may easily be extended to support others. Unlike TOPCASED [19], TTool does not use a meta-modeler, nor it is linked to Eclipse, which makes it independent from any third-party tool developer. As far as verification is concerned, TTool does not rely on OCL [91] but TEPE as property expression language. Further, the current version of TTool, as well as ongoing developments, aims to make the use of temporal logic and formal languages transparent to users.

3.5.4 Conclusions

As the objective is to verify sequential behavior and its timing, property descriptions could rely on state machines. However, if TEPE is applied to system models based on UML Statecharts, the same formalism would be employed both for design and verification. This is definitely not a good choice, as verification runs the risk of being hampered by the same errors in reasoning as the model. Moreover, (1) the number of states tends to explode when events may be received in various orders, and (2), statecharts put emphasis on implementation rather than on property relations, like TEPE. As mentioned in 3.5.2 formally defined descriptions for sequential behavior fall short in UML.

To conclude, the comparison to existing work exposes the following strengths of TEPE, which . . .

1. . . . focuses on relations of properties
2. . . . supports both state and event-based views of a system

3. . . . allows to seamlessly accomplish design and verification in UML
4. . . . relies on different formalisms than system design
5. . . . is more intuitive than temporal logic while exhibiting a comparable expressiveness
6. . . . provides a means to express time constraints
7. . . . applies directly to all Models of Computation exhibiting states and signals
8. . . . distinguishes safety and liveness properties
9. . . . generates helper signals and attributes that emulate non observable transitions and states in the system model.

3.6 Modeling and visualization

For the construction of models and the visualization of simulation progress, we use an in-house UML tool called TTool. On the one hand, it has a longer history than most of the state of the art tools and has proven to be of great value in various projects and collaborations with academia and industry. On the other hand, it comes with the advantage of not being dependent on a third party, especially when bugs are encountered. Moreover, animation of UML diagrams and direct feed-back of simulation results requires well defined interfaces between modeling and simulation facilities. The latter are far easier to achieve in an approach from one source.
TTool's new capabilities are examined in more detail in Chapter 7. Some of the current state of the art UML modeling tools (Topcased [120], Tau [119], Rhapsody [106], Artisan [18] amongst others) provide simulation features. Simulations can only be performed based on purely functional models in an untimed fashion. Our interactive simulator however also accounts for architecture semantics such as arbitration of shared resources, speed or data throughput of devices, etc. Furthermore, the execution behavior of models is tool dependent as the UML standard lacks an execution semantics. The DIPLODOCUS profile however fills that semantic gap and thus also paves the way for formal verification.

3.7 Conclusions

Nowadays, the separation of concerns, namely application and architecture, has been seized by several academic and industrial approaches and is referred to as the Y-Chart approach. However, there are several shortcomings of related work which have already been addressed by our environment or which are dealt with in the scope of this thesis:

- Purely analytical (data flow) models tend to be overly pessimistic or optimistic (WCET BCET) and do not detail control flow within tasks at all. Our framework however allows for detailing task behavior in accordance with the aspired level of abstraction. The metamodel is endowed with a formal semantics and thus paves the way for simulation and formal verification combined in the same framework.

- In some frameworks, models need to be refined (using code) before being simulated which entails an increased modeling effort and badly affects simulation performance. Our solution generates executable models at the push of a button, without requiring any expertise in simulation or formal proofs techniques.

- Almost all simulation frameworks rely on SystemC and thus tend to make use of detailed architecture models which are very costly in terms of simulation time. Furthermore, the standard SystemC simulation kernel is not really vaunted for its efficiency. Making extensive use of signals and processes slows down the simulation due to an inefficient scheduling and results in unnecessary process wake-ups. The seemingly promising but dangerous advantage is the compatibility to existing low level models which could be used for Design Space Exploration purposes. We believe that the key to efficient DSE is a meta model introducing abstractions which are leveraged by the simulation environment.

- Many employed MoCs (KPN, data flow networks,..) do not make data dependent control flow variations or indeterminism explicit at application level. Implicitly, analysis merely covers an average load of the system, which does not necessarily reflect realistic conditions.

- To our knowledge, state of the art UML model simulators target a solely functional verification of the system. Semantics and constraints of shared resources and more precisely communication and computation architectures are not considered at all. In the scope of this thesis, a framework is elaborated which is capable of animating UML application models while being executed on a specific architecture configuration.

- The additional modeling effort inherent to some other approaches is mitigated in our case by the usage of UML models and automated model transformations. The necessary UML models can be regarded as a specification which would have had to be established anyway, but with the positive side-effect of being executable and analyzable.

- Many environments do not fathom the trade-off between exhaustive formal and simulation techniques. Verification is thus hampered by state explosion or a too narrow coverage of the application. In DIPLODOCUS, an explicit control flow representation comprising operators for indeterminism enable a variable coverage of the application model.

- Even though some frameworks are settled at a high level of abstraction, verification relies on obscure logical formulas or instrumentation of source code with assertions. A seamless integration of verification techniques into the design flow is not provided.

After having reviewed shortcomings of state-of-the art approaches, this report now elaborates on the contributions made in the field of verification, requirement capture, simulation and tooling.

Chapter 4

TEPE - A formal, graphical verification language

4.1 Introduction

The verification of abstract system level models is still hampered by the required expertise in temporal logic. Designers less familiar with that domain would definitely appreciate a verification language that matches the abstraction level of the model to be verified. To address this issue, we advocate TEPE[1], a user-friendly graphical TEmporal Property Expression language formally defined with fluents. TEPE both supports state-based and event-based formalisms. It applies to a broad variety of systems defined in terms of states (attributes) and transitions (signals) between these states.

The increasing importance of real-time systems in life-critical applications has stimulated research work on modeling techniques that combine the friendliness of UML / SysML with the formality of verification tools such as Temporal Logic, UPPAAL [24], etc. So far, the use of SysML in verification centric methods has been hampered by the poor formality of Requirement Diagrams and the lack of powerful property expression languages. Thus, UML profiles commonly require the use of temporal logic (e.g., CTL) that might not meet the level of abstraction of the system model. The MARTE standard embraces VSL [92] which is more concerned with the specification of values related to non-functional aspects. When it comes to the verification of sequential behavior, MARTE suggests the Clock Constraint Specification language (CCSL). This theoretical framework has a solid mathematical foundation but leaves the abstraction of system transitions to clocks and practical tooling issues to the designer. Users lacking a mathematical background may be discouraged by formal methods not featuring a friendly visualization.

To address the aforementioned shortcomings, this work extends SysML Parametric Di-

[1] TEmporal Property Expression Language

agrams with TEPE, a graphical but formal language for describing logical and temporal properties. In TEPE, various design elements, such as SysML blocks attributes and signals, can be combined together with logical (e.g., sequence of signals) and temporal operators (e.g., a time interval for receiving a signal) to build up complex graphical properties.
TEPE offers an intuitive two-dimensional way to compose safety and liveness properties built upon constraints. The language comes with a graphical front-end that originates from SysML parametric diagrams. Moreover, TEPE could be introduced into the OMG-based SysML and a broad variety of SysML profiles. To provide evidence for this, we integrate TEPE into two UML/SysML environments: AVATAR[1] and DIPLODOCUS. Thanks to the joint use of TEPE and an UML/SysML environment, requirement capture, analysis, design, property description and verification tasks can seamlessly be accomplished in the same language with the same tool (cf. chapter 7). To utilize TEPE, the designer is merely required to have minor UML skills and does not need any expertise in formal languages such as CTL or UPPAAL.

The chapter is organized as follows. Section 4.2 surveys Metric Temporal Logic and Fluent Temporal Logic serving as a formal base for TEPE constraints. Section 4.3 introduces the TEPE language both in an intuitive and a formal way. Finally, two sections are devoted to the integration of TEPE into two representatives of UML/SysML profiles namely the AVATAR (section 4.4) and DIPLODOCUS (section 4.5). A joint case study in DIPLODOCUS and AVATAR is presented covering design, property modeling and verification stages of a microwave oven. Finally, Section 4.6 concludes this chapter.

4.2 Formal toolbox

This section surveys two logics used to formalize TEPE later in this chapter. The combination of Metric Temporal Logic (MTL) and Fluent Linear Temporal Logic (FLTL) is especially useful for reasoning about temporal properties in terms of both system states and events.

4.2.1 Metric Temporal Logic (MTL)

Given a set of atomic propositions Π, a state trace $h : \mathbb{N} \rightarrow 2^{\Pi}$ maps to each position $i \in \mathbb{N}$ the set of propositions that hold at that position. Metric Temporal Logic (MTL) is presented in [67] and enriches LTL with bounded temporal operators: $[]_{\sim d}\ P$, $<>_{\sim d}\ P$ and $P\ U_{\sim d} Q$ where $\sim \in \{<, \leq, >, \geq\}$ and $d \in \mathbb{N}$. Therein, P stands for an MTL formula, [] and <> for the LTL operators *always* and *eventually*. MTL is defined with respect to a temporal distance function $dist : \mathbb{N} \times \mathbb{N} \rightarrow T$, where T is the time domain and $dist\,(i, j)$ denotes the time elapsed between position i and j in a trace. The distance function is

[1] Automated Verification of reAl Time softwARe

required to exhibit the properties of a metric. For this work, only the *always* and the *eventually* operators are relevant. The notation $(h, i) \models P$ means that the MTL formula P is true at position i of trace h.

- $(h, i) \models []_{\sim d} P$ iff $(h, j) \models P$ for all $j \geq i$ and $dist\,(i, j) \sim d$
- $(h, i) \models <>_{\sim d} P$ iff $(h, j) \models P$ for at least one $j \geq i$ and $dist\,(i, j) \sim d$

4.2.2 Fluent Linear Temporal Logic (FLTL)

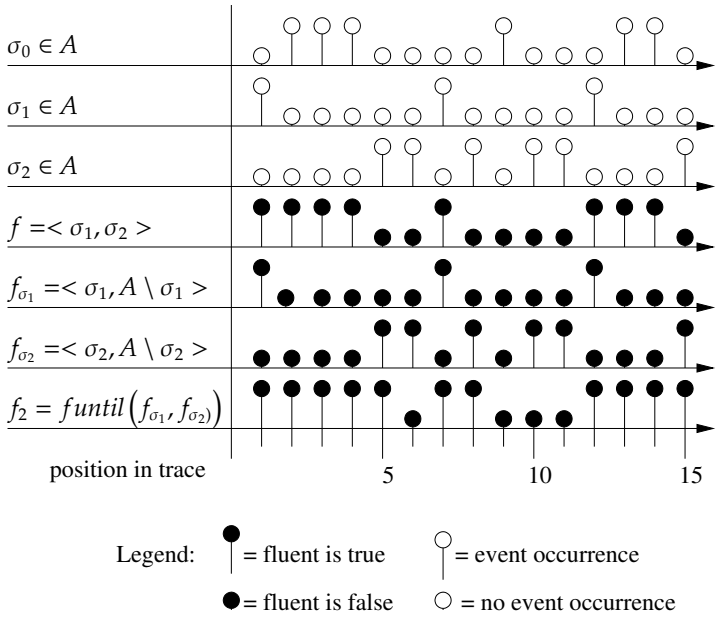


Figure 4.1: Fluent example

Fluents are defined in [75] as "state predicates whose values are determined by the occurrence of initiating and terminating events". A fluent is thus a two tuple comprising an initiating event σ_1 and a terminating event $\sigma_2 \neq \sigma_1$ and is initially defined to be false[1]. Fluents are denoted as suggested in [75]:

[1] In [75], fluents are defined over sets of initiating and terminating events and the initial value can be set arbitrarily. For our purposes, the simplified definitions are sufficient.

$f =< \sigma_1, \sigma_2 >$. Let A be the set of all distinct events contained in an event trace $tr : \mathbb{N} \rightarrow A$. A set Φ of fluents f maps an event trace tr to a corresponding state trace $h = StateTrace(tr)$ which is defined as follows: for every position $i \in \mathbb{N}$ and every fluent $f \in \Phi$, f is true at position i of h iff the following conditions are satisfied:

- Informally, some initiating event must have occurred before position i and no terminating event has occurred since then.
- There exists some $j \in \mathbb{N}$, $j \leq i$ such that $tr(j) = \sigma_1$ and there is no $k \in \mathbb{N}$, $j < k \leq i$ such that $tr(k) = \sigma_2$.
- An FLTL assertion P is said to be satisfied by an event trace tr, iff $StateTrace(tr)$ satisfies P.

By default, the interval over which a fluent is true is closed on the left and open on the right. This means that an event at position i in the event trace affects a fluent starting from the same position, namely i. This can be seen in Figure 4.1, where the fluent $f =< \sigma_1, \sigma_2 >$ becomes true at the instant of occurrence of signal σ_1. In the same way, the occurrence of sigma σ_2 has an immediate impact on f; the latter becomes false right away.
In analogy to [75], we also define fluents for event occurrences. For an event σ, the corresponding fluent is terminated by any other event in the system alphabet, and hence $f_\sigma =< \sigma, A \setminus \sigma >$. The fluent holds at the instant when the event occurs and becomes false upon the first occurrence of any other event (cf. f_{σ_1} and f_{σ_2} in Figure 4.1)
Informally, the function **funtil**: $\Phi \times \Phi \rightarrow \Phi$ derives a fluent $f_1 = funtil(f_2, f_3)$, which is true at position $i \in \mathbb{N}$ if f_2 was true at a position before i and f_3 has not been true since then. Hence, f_1 is true at position i if there is some $j \in \mathbb{N}$, $j \leq i$ where $f_2(j) = true$ and there is no $k \in \mathbb{N}$, $j < k < i$, where $f_3(k) = true$. The meaning of $funtil$ can be understood by looking at Figure 4.1: f_2 is true from the instant when f_{σ_1} is true until the instant when f_{σ_2} is true, including the boundaries.
Finally, two special cases of fluents are given a name: the **false-fluent** f_{false} is always false and the **true-fluent** f_{true} is always true. To simplify textual descriptions, we define an **equivalent signal** σ of a fluent f as a signal which is notified at position i in an event trace tr if f is true at that position in the state trace h: $\sigma = sigequiv(f)$. In Figure 4.1 for example, the following two identities hold: $\sigma_1 = sigequiv(f_{\sigma_1})$ and $\sigma_2 = sigequiv(f_{\sigma_2})$.
Boolean algebra over fluents such as $f_r = f_1\ OP\ f_2$ is simply defined as an element-wise application $f_r(i) = f_1(i)\ OP\ f_2(i)$ of the operator OP. For a given trace tr, all fluents are supposed to have the same length as the trace.

4.3 TEPE: TEmporal Property Expression language

4.3.1 Requirements modeling with SysML Requirement Diagrams

SysML Requirement Diagrams (RDs) establish relations among *requirements* and define *testcases*. Examples of these relations are << *deriveReqt* >> and composition relations. Requirements may also be copied from other views (<< *copy* >>). SysML RDs define the notion of testcases (named "*properties*" in the following) that are linked to requirements using the << $verify$ >> relation. Unfortunately, in SysMLRDs, properties are only informally specified with an identifier and a plain text. The following section suggests TEPE to remedy this shortcoming.

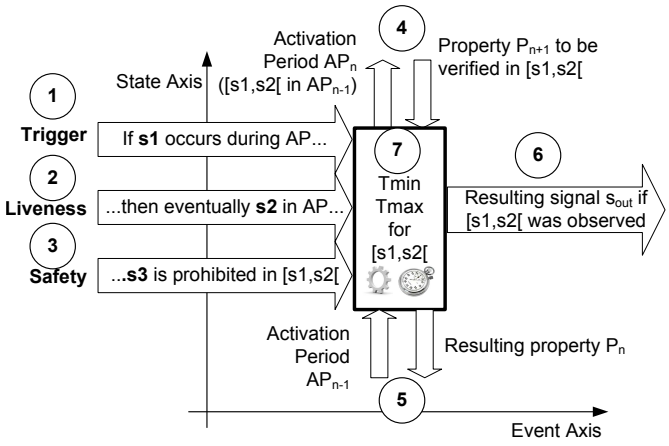
4.3.2 Parametric Diagrams

4.3.2.1 Intuition

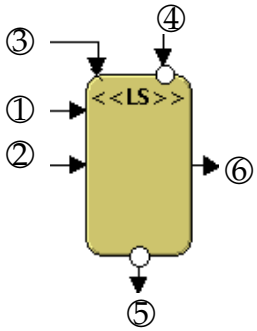
TEPE constraints directly refer to attributes and signals, that is system states and events in the MoC terminology. MoCs exhibiting either of the two primitives are amenable to verification with TEPE. That way, TEPE acknowledges the fact that properties are sometimes more conveniently formulated in either a state-based or an event-based fashion. As a rule of thumb, whenever a system's history is relevant for a property, a state-based expression is preferable. If a property applies in very different scenarios, which are characterized by common behavioral patterns, then events are the formalism of choice.

TEPE was built on the two simple insights that properties are often not invariants and that liveness and safety properties should be easily expressible and distinguishable. As depicted in Figure 4.2a, vertically cascaded TEPE constraints determine the *activation period* AP_n of the constraint immediately above. This is suggested by the vertical AP_n arrow marked with ④. In turn, they only operate on signals and properties during the activation period AP_{n-1} specified by the constraints immediately below (arrow ⑤). Properties, which are state predicates, are propagated along the vertical axis (cf. *property arrows* P_{n+1} and P_n in the figure), named state axis. A constraint receives an input property P_{n+1} to be verified and produces an output property P_n, which may be the final result or subject to further verification. During its activation period AP_{n-1}, a constraint observes three signals represented by horizontal arrows in Figure 4.2a. The signal $s1$ (arrow ①) serves as **trigger** for the verification of the **liveness** of $s2$ (arrow ②), and the **safety** (arrow ③) of the system prohibiting the occurrence of $s3$ between $s1$ and $s2$. To account for time constraints, TEPE operators may specify a minimum **duration** T_{min} and maximum duration T_{max} (marked with ⑦) of the interval bounded by $s1$ and $s2$ (marked with ⑦ as well). TEPE operator may also generate an output signal, indicating that an input sequence has been correctly observed (arrow ⑥)
The UML view below the intuitive representation reveals that **TEPE operators** only **support a subset of the** aforementioned **features**. For instance, the *Sequence Constraint*

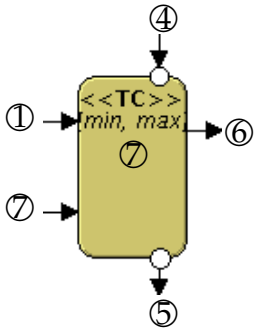
(LS) in Figure 4.2b has a trigger signal $s1$ and monitors the liveness of signal $s2$ and the non-occurrence of $s3$ during the period bounded by $s1$ and $s2$. However, the time that elapses during that period cannot be constrained, this can exclusively be realized with the *Temporal Constraint* (TC) in Figure 4.2c. The latter does not offer signal inputs for liveness and safety checks. Figure 4.2a further shows that signals are symbolized by normal arrows (⟶), whereas properties arrows (→○) are marked with a circle at both ends. Property arrows both symbolize a state predicate named P_{n+1}/P_n in the intuitive schema, and the activation period AP_n/AP_{n-1}.
In subsequent sections, we will formalize how incoming arrows in Figure 4.2a ($s1$, $s2$, $s3$, AP_{n-1}, P_{n+1}) relate to outgoing arrows (P_n, s_{out}, AP_n) by means of the **transfer functions** F_{prop}, F_{sig}, F_{act}. Therefore, **signals and activation periods will be abstracted to fluents**, and properties will be deduced as a result of an expression in FLTL.



(a) Intuition for the TEPE semantics



(b) Sequence Constraint, UML based notation



(c) Time Constraint, UML based notation

Figure 4.2: Intuition and corresponding UML based notations

4.3.2.2 Construction

Since SysML Parametric Diagrams (PDs) establish constraints between elements of system design - i.e., parameters -, TEPE introduces a small set of SysML Constraints called TEPE Contraints, or *TC* for short. As opposed to informal SysML PDs, TEPE PDs are amenable to automated verification.

In TEPE, each property is expressed as a graph of TEPE Constraints which are: *Alias Constraints*, *Conjunction Constraints*, *Disjunction Constraints*, *Equation Constraints*, *Logical Constraints*, *Sequence Constraints*, *Temporal Constraints*, *Property Definition Constraints* and *Setting Constraints*. Constraints are related to each other using three types of parameters: *Signals*, *Attributes* and *Properties*. An excerpt of the TEPE meta model is depicted in Figure 4.3. All stereotypes of PDs are derived from their respective SysML counterpart: *Blocks*, *Constraints* and *Links*. A block provides a scope to *Attributes* and *Signals* in order to unambiguously map them to the system model. Properties are assembled by means of *Links* which are attached to the ports of *Constraints*. *Links* may be established between two parameters of the same type (Attribute, Signal or Property). *Ports* must obviously have the same data type as the connected *Link*, and two connected ports must have the inverse input/output configuration.

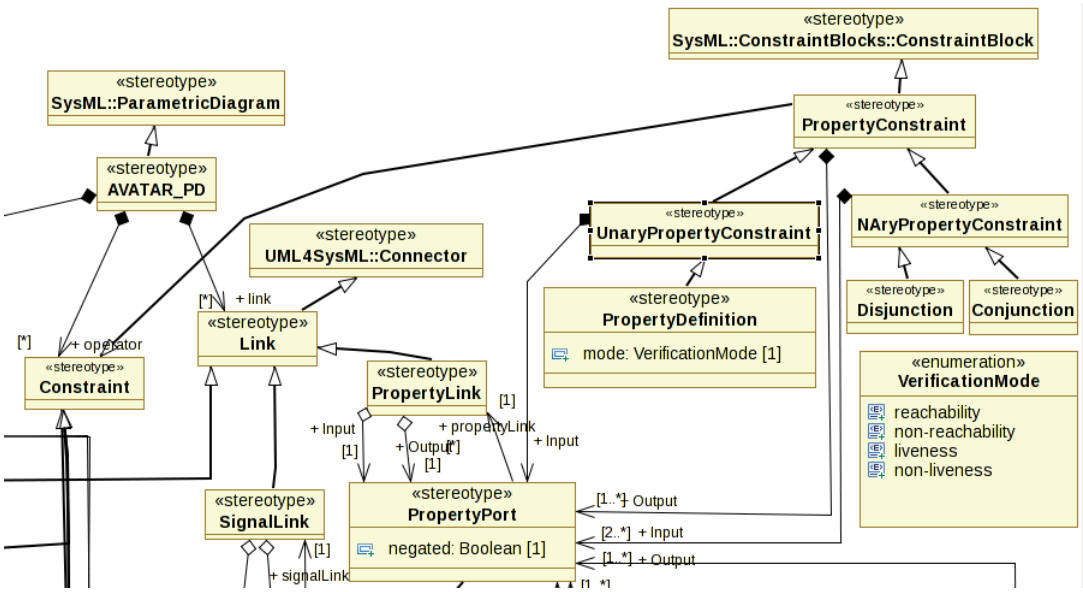


Figure 4.3: Excerpt from the TEPE PD Meta Model

A TEPE PD can be constructed in the following way:

1. First, *Blocks* are represented with their particular *Attributes* and *Signals* subject to verification. These entities have been identified during the design phase. A *Block* may refer to any entity in the system model providing a scope to *Attributes* and *Signals*.

2. Values derived from original attributes and signals are introduced (cf. *Equation* and *Alias* constraints).

3. The reasoning about the sequential and temporal traces of the system is expressed in terms of logical and temporal constraints. These constraints can be composed using *Signals* and *Property* parameters.

4. Several *Properties* may be combined via logical property constraints such as *Conjunction*, *Disjunction* and *Property Definition* constraints.

5. Finally, using a reference label, the formal property to be verified is linked to an informal testcase of a SysML RD. The formal property is tagged with a quantifier attribute (non-) liveness or (non-) reachability.

6. To avoid overloaded diagrams, (possibly multiple) properties of a requirement can be spread over several diagrams.

4.3.2.3 Example

The example in Figure 4.4 informally presents operators of PDs, which are described in section 4.3.3 in more detail. The represented properties are somewhat artificial, but nicely demonstrate the basic concepts of TEPE. More realistic properties can be found in section 4.5.2.2 and 8.3.1.4.

- In the interval starting with $s3$ and ending with $s1$ or $s2$, the equation $z > 0$ must be satisfied, and the expression $z = x + y$ must remain constant. Moreover, the interval must finally be concluded with $s1$ or $s2$.

- The signal $s2$ must be observed less than 10 time units after signal $s1$.

- We finally express that both properties have to hold.

The depicted PD defines two Blocks: *BlockA* has two attributes x and y as well as two signals $s1$ and $s2$. *BlockB* declares a signal called $s3$. A *Setting* constraint declares a temporary variable $z = x + y$ which simply serves as a shorthand to derive other expressions. The equation $z > 0$ is connected to the *Logical Sequence* (LS) operator and is therefore only verified during its activation period, specified by the *Sequence Constraint* (LS). An *Alias* constraint combines the two signals $s1$ and $s2$. The resulting signal is raised upon occurrence of either of the two input signals $s1$ or $s2$. The two properties, established by the *LS* and *TC* constraint, are combined with an *AND* constraint. The *LS* constraint requires that upon occurrence of an $s3$ signal, the compound signal resulting from the *Alias* constraint must eventually be observed, i.e. $s1$ or $s2$. Note that the safety input of the *LS* constraint is connected to the toggle signal of the equation. In case of an equation, a toggle signal announces a change from false to true, in case of a setting each value change of the expression triggers an occurrence of the toggle signal. Therefore, if the value of z changes or if the equation $z > 0$ is not satisfied between the occurrence of $s3$

and the compound signal, the LS constraint evaluates to false. The property established by the TC constraint requires the signal *s*2 to occur *less than 10 time units after* signal *s*1. The property resulting from the *AND* constraint must be satisfied for every execution. This is made explicit with a *Property Definition* constraint configured for the verification of liveness.

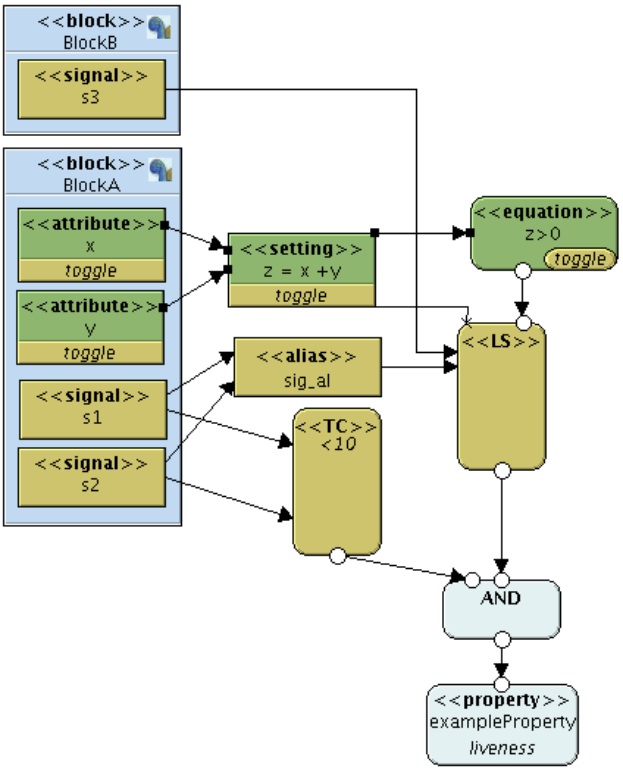


Figure 4.4: Example of a TEPE Parametric Diagram

4.3.3 Links

TEPE constraints are horizontally connected together with *attribute* and *signal* links, and vertically with *property* links.

- →■ **Attribute** links refer to the system model when originating from a *Attribute declaration* constraint, or to attributes introduced withing a TEPE PD when originating from a *Setting Constraint*. The only purpose of attribute links is to unambiguously identify an attribute, even if several attributes of the same name exist. In UML, the arrows of attribute links are decorated with a small rectangle at both ends.
- → **Signal** links convey signals originating directly from the system model, or signals resulting from TEPE constraints such as toggle signals, *Alias* signals, etc.

To visually separate the state and signal domain, signals are connected to the left and right border of constraints.

- →○ **Property** links convey state predicates, thus boolean values resulting from *Equation*, *Temporal*, *Sequence* or *Logical* constraints. Properties represent the state domain in TEPE and are connected to the top and bottom border of TEPE constraints. In the following, constraint $c1$ providing a property to be verified to another constraint $c2$ is said to be above $c2$. Conversely, constraint $c2$ verifying this property is below $c1$. In UML, the arrows of property links are decorated with with a small circle at both ends.

4.3.4 Generic TEPE Constraints

The formal definition of a TEPE constraint allows us to express each output of a TEPE constraint as a function of its inputs. These so called transfer functions are formulated for each constraint in FLTL. The following definition is generic in the sense that it embraces the superset of all inputs and outputs of all types of TEPE constraints. For the existing constraints however, only a fraction of these elements are relevant.
A generic TEPE constraint is an 9-tuple (F_i, F_{neg}, F_{out}, T_{par}, F_{sai}, F_{ao}, $\mathbb{F}_{prop}$, $\mathbb{F}_{sig}$, $\mathbb{F}_{act}$) comprising

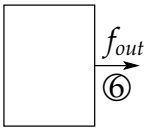
- A set F_i of input fluents, with at most two input fluents f_{i1}, f_{i2}
- A set F_{neg} of negated fluents, with at most one negated fluent f_{neg}
- A set F_{out} of output fluents, with at most one output fluent f_{out}
- A set T_{par} of time parameters, with at most two time parameters denoting a minimum and a maximum time and referred to as T_{min} and T_{max} respectively
- A set F_{sai} with at most one pair of an activation period input and a property output: $F_{ai} = \{f_{n-1}, P_n\}$
- A set of at most two pairs of an activation period output and a property input: $F_{ao} = \{\{f_{n,1}, P_{n+1,1}\}, \{f_{n,2}, P_{n+1,2}\}\}$. To simplify matters, $\{f_{n,1}, P_{n+1,1}\}$ is written as $\{f_n, P_{n+1}\}$ if there exists exactly one pair.
- A transfer function for the property output if applicable $P_n = \mathbb{F}_{prop}\left(F_i,\ F_{neg},\ \left\{P_{n+1,1} \ldots P_{n+1,j}\right\},\ f_{n-1},\ T_{par}\right)$, expressed in FLTL.
- A transfer function for the output fluent if applicable $f_{out} = \mathbb{F}_{sig}\left(F_i,\ f_{n-1}\right)$, expressed in FLTL.
- A transfer function for the activation period output if applicable $\left\{f_{n,1} \ldots f_{n,j}\right\} = \mathbb{F}_{act}\left(F_i,\ f_{n-1}\right)$, expressed in FLTL.

In the further course of the formalization, we will assume that the activation period of the lowermost constraint $f_{n-1,low}$ in a set of vertically cascaded TEPE constraints is the true-fluent $f_{n-1,low} = f_{true}$.

4.3.5 Attribute constraints

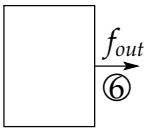
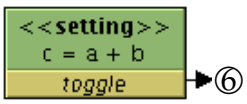
Attribute constraints make system state variables, called attributes, amenable to verification with TEPE. The respective attributes must have been declared in the system model or in *Settings*. As stated in the introduction, properties may sometimes be more conveniently expressed in a signal based manner. If the system model does not associate an observable signal to a change of an attribute, this is where toggle signals come in. Toggle signals are defined to be sent upon a change of an expression depending on system attributes, such as settings, equations or single attributes. That way, these signals bridge the gap between the state and the signal domain.

4.3.5.1 Attribute Declaration

Semantic view	*Signal/Fluent relations*	*UML view*
	$\sigma_{out} = sigequiv\,(f_{out})$	

An Attribute declaration refers to an attribute defined in the system model. The toggle signal σ_{out} is sent whenever the attribute changes its value.

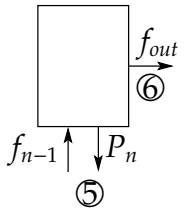
4.3.5.2 Setting

Semantic view	*Signal/Fluent relations*	*UML view*
	$\sigma_{out} = sigequiv\,(f_{out})$	

Settings declare a new attribute as a function of existing ones. These attributes may be useful to simplify otherwise complex expressions. The toggle signal σ_{out} is sent whenever the declared attribute changes its value.

4.3.5.3 Equation

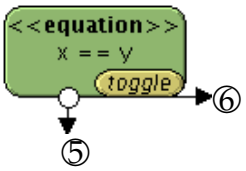
Semantic view



Signal/Fluent relations

$\sigma_{out} = sigequiv\,(f_{out})$

UML view



An Equation constraint consists of a boolean expression Eq which is a function of attributes. Eq has to hold during the activation period for the output property P_n to evaluate to true. The toggle signal σ_{out} is sent upon a change of the equation result from false to true. The transfer function is:

$$\mathbb{F}_{prop}: \; P_n = (f_{n-1} \rightarrow Eq)$$

4.3.6 TEPE Signal constraints

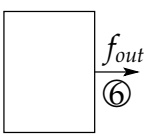
Signal constraints express a property in terms of the presence and the absence of signals during the activation period f_{n-1}. As the semantic view suggests, *Logical*, *Sequence* and *Time Constraints* represent the intersection of TEPE's signal and property axes. The activation period of a signal constraint is determined by the operator below and the given constraint in turn determines the activation period of the operator above. Depending on the constraint type, the interval denoted by two signals σ_1 and σ_2 may be required ...

- ... not to include a signal σ_{neg}
- ... to satisfy a property P_{n+1}
- ... to terminate with the signal σ_2 before the end of the activation period f_{n-1}
- ... not to exceed a maximum duration t_{max}; not to fall below a minimum duration t_{min}

for the property to be satisfied.

4.3.6.1 Signal declaration

Semantic view



Signal/Fluent relations

$\sigma_{out} = sigequiv\,(f_{out})$

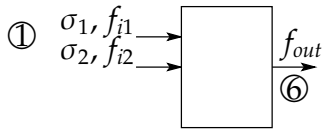
UML view

A signal declaration merely makes a signal $\sigma_{out} \in A$ occurring in a system's event trace tr available for TEPE constraints by converting it into a fluent f_{out}. Its semantics unambiguously relates input fluents of TEPE constraints to signals of the system model.

4.3.6.2 Signal Alias

The output signal σ_{out} of an Alias constraint is a signal notified whenever either of the two input signals σ_1 and σ_2 occurs. The constraint can be regarded as a logical disjunction in the signal domain.

Semantic view



Signal/Fluent relations

$$\sigma_1 = sigequiv\,(f_{i1})$$
$$\sigma_2 = sigequiv\,(f_{i2})$$

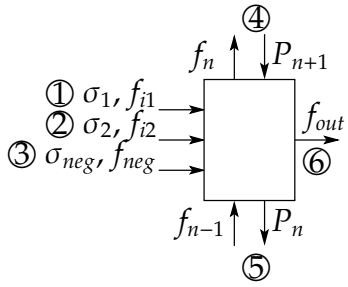
UML view

The transfer function is:

$$\mathbb{F}_{sig}:\ f_{out} = f_{i1} \vee f_{i2}$$

4.3.6.3 Sequence Constraint

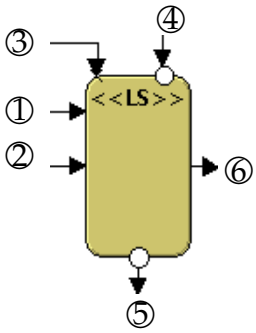
Semantic view



Signal/Fluent relations

$$\sigma_1 = sigequiv\,(f_{i1})$$
$$\sigma_2 = sigequiv\,(f_{i2})$$
$$\sigma_{neg} = sigequiv\left(f_{neg}\right)$$

UML view

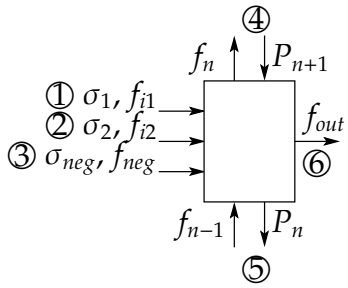


The activation period of the constraint is determined by the constraint below and referred to as f_{n-1}. Informally, if, during an activation period, the occurrence of signal σ_1 is eventually followed by the occurrence of signal σ_2, and if during the interval delimited by σ_1 and σ_2, the negated signal σ_{neg} is not observed and the property P_{n+1} holds, the property P_n evaluates to true. Otherwise, P_n is defined to be false. During the activation period, the output signal σ_{out} is notified upon an occurrence of σ_2, if it was preceded by an occurrence of σ_1. The transfer functions are:

$$\mathbb{F}_{prop}:\ P_n = (f_{n-1} \wedge f_{i1}) \rightarrow X\left(\left(\neg f_{neg} \wedge P_{n+1} \wedge f_{n-1}\right)\ U\ f_{i2}\right)$$
$$\mathbb{F}_{act}:\ f_n = f_{n-1} \wedge funtil\,(f_{i1}, f_{i2})$$
$$\mathbb{F}_{sig}:\ f_{out} = f_n \wedge f_{i2}$$

4.3.6.4 Logical Constraint

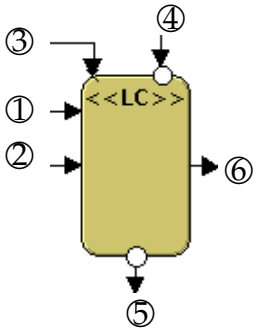
Semantic view



Signal/Fluent relations

$$\sigma_1 = sigequiv\,(f_{i1})$$
$$\sigma_2 = sigequiv\,(f_{i2})$$
$$\sigma_{neg} = sigequiv\left(f_{neg}\right)$$

UML view

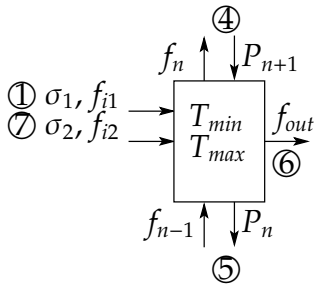


The Logical Constraint extends the rules of Sequence Constraint (marked with (a) in the following) with the same rules for the reverse signal order (marked with (b) in the following). The activation period of the constraint is determined by the constraint below and referred to as f_{n-1}. Informally, if, during an activation period, the occurrence of signal σ_1(a)/σ_2(b) is eventually followed by the occurrence of signal σ_2(a)/σ_1(b), and if during the interval delimited by σ_1(a)/σ_2(b) and σ_2(a)/σ_1(b), the negated signal σ_{neg} is not observed and the property P_{n+1} holds, the property P_n evaluates to true. Otherwise, P_n is defined to be false. During the activation period, the output signal σ_{out} is notified upon an occurrence of σ_2(a)/σ_1(b), if it was preceded by an occurrence of σ_1(a)/σ_2(b). Thereby, both signals σ_1 and σ_2 exclusively belong to one interval, and either open or close it, but not both. For example, the sequence of signals σ_1, σ_2 complies to the rules marked with (a). However, to switch to case (b), the sequence must be followed by a second occurrence of σ_2. The transfer functions are:

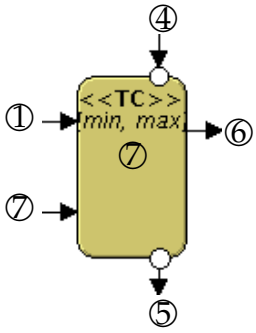
$$\mathbb{F}_{prop}:\ P_n = \Big((f_{n-1} \wedge f_{i1} \wedge \neg f_{o2}) \rightarrow X\Big(\big(\neg f_{neg} \wedge P_{n+1} \wedge f_{n-1}\big)\ U\ f_{i2}\Big)\Big) \wedge$$
$$\Big((f_{n-1} \wedge f_{i2} \wedge \neg f_{o1}) \rightarrow X\Big(\big(\neg f_{neg} \wedge P_{n+1} \wedge f_{n-1}\big)\ U\ f_{i1}\Big)\Big)$$
$$\mathbb{F}_{act}:\ f_n = f_{n-1} \wedge (f_{o1} \vee f_{o2})$$
$$\mathbb{F}_{sig}:\ f_{out} = f_n \wedge ((f_{o1} \wedge f_{i2}) \vee (f_{o2} \wedge f_{i1}))$$
where $f_{o1} = funtil\,(f_{i1}, f_{i2})$ and $f_{o2} = funtil\,(f_{i2}, f_{i1})$

4.3.6.5 Temporal Constraint

Semantic view | *Signal/Fluent relations* | *UML view*



$\sigma_1 = sigequiv(f_{i1})$
$\sigma_2 = sigequiv(f_{i2})$



Depending on the supplied number of signals and parameters, the Temporal Constraint has several semantics. In case it operates on two signals σ_1 and σ_2, both physical times t_{min} and t_{max} or only one of the two may be specified. In case the Temporal Constraint operates on one signal σ_1, exactly one time value T must be provided. In the following, the operating modes and their corresponding transfer function are summarized.

1. $\sigma_1, \sigma_2, t_{min}, t_{max}$ supplied:
   σ_2 has to occur at least t_{min} time units, at most t_{max} time units after σ_1 and P_{n+1} must be satisfied from the occurrence of σ_1 until the occurrence of σ_2 (Figure 4.5a). The transfer functions are:

   $$\mathbb{F}_{prop}: P_n = (f_{n-1} \wedge f_{i1}) \rightarrow (([]\{< T_{min}\}\neg f_{i2}) \wedge (<> \{\leq T_{max}\} f_{i2}) \wedge (X(P_{n+1}\ U\ f_{i2})))$$
   $$\mathbb{F}_{act}: f_n = f_{n-1} \wedge funtil(f_{i1}, f_{i2})$$
   $$\mathbb{F}_{sig}: f_{out} = f_n \wedge f_{i2}.$$

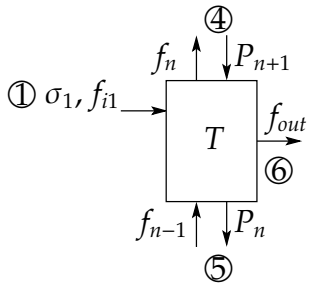
2. $\sigma_1, \sigma_2, t_{max}$ supplied:
   σ_2 has to occur at most t_{max} time units after σ_1 and P_{n+1} must be satisfied from the occurrence of σ_1 until the occurrence of σ_2 (Figure 4.5b). The transfer functions are:

   $$\mathbb{F}_{prop}: P_n = (f_{n-1} \wedge f_{i1}) \rightarrow ((<> \{\leq T_{max}\} f_{i2}) \wedge (X(P_{n+1}\ U\ f_{i2})))$$
   $\mathbb{F}_{act}$ and $\mathbb{F}_{sig}$ are defined as above.

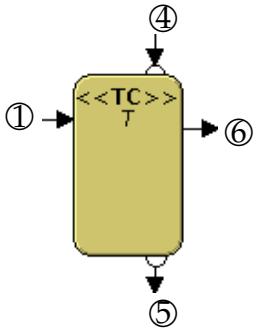
3. $\sigma_1, \sigma_2, t_{min}$ supplied:
   σ_2 has to be notified at least t_{min} after σ_1 and P_i must be satisfied from the occurrence of σ_1 until the occurrence of σ_2 (Figure 4.5c). The transfer functions are:

   $$P_n = (f_{n-1} \wedge f_{i1}) \rightarrow (([]\{< T_{min}\}\neg f_{i2}) \wedge (X(P_{n+1}\ U\ f_{i2})))$$
   $\mathbb{F}_{act}$ and $\mathbb{F}_{sig}$ are defined as above.

Semantic view *Signal/Fluent relations* *UML view*

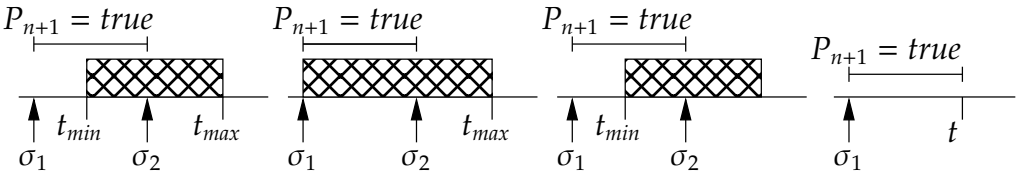


$\sigma_1 = sigequiv\,(f_{i1})$



4. σ_1, t supplied:
   after occurrence of σ_1, P_{n+1} must be satisfied at least for t time units (Figure 4.5d).
   The transfer functions are:

$\mathbb{F}_{prop}: \; P_n = (f_{n-1} \wedge f_{i1}) \rightarrow ([]\{< T\}P_{n+1})$
$f_{to} =< \sigma_{to}, A \setminus \sigma_{to} >$ where σ_{to} indicates the expiration of t time units
$\mathbb{F}_{act}: \; f_n = f_{n-1} \wedge funtil\,(f_{i1}, f_{to})$
$\mathbb{F}_{sig}: \; f_{out} = f_{to}$



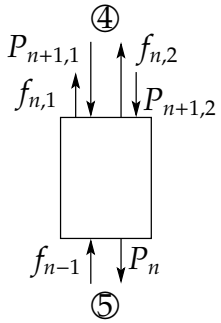
(a) Semantics 1 (b) Semantics 2 (c) Semantics 3 (d) Semantics 4

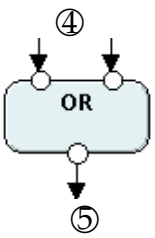
Figure 4.5: Temporal Constraint Operator Semantics

4.3.7 Property Constraints

4.3.7.1 Property Logic

Semantic view *UML view*



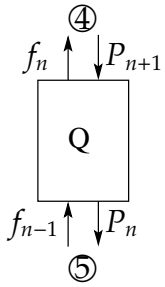


Property Logic Constraints perform logical operations *OP* such as conjunction and disjunction on properties. The transfer functions are:

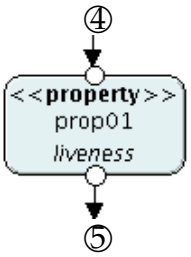
$$\mathbb{F}_{prop} : P_n = P_{n+1,1}\ OP\ P_{n+1,2}$$
$$\mathbb{F}_{act} : \{f_{n,1}, f_{n,2}\} = \{f_{n-1}, f_{n-1}\}$$

4.3.7.2 Property Label

Semantic view



UML view



Property Labels assign a name to a TEPE property and provide path and temporal quantifiers. So far, three combined quantifiers Q have been predefined: *Reachability*, *Liveness* and *Safety*, which are equivalent to the CTL semantics of *EF*, *AF* and *AG* respectively. From our experience, these three cases already cover the lion's share of usual verification scenarios. However, path and temporal quantifiers could also be freely combined by the user in future versions of TEPE, if it turns out that the expressiveness is limited by our choice. The transfer functions are:

$$\mathbb{F}_{prop} : P_n = Q\ P_{n+1}$$
$$\mathbb{F}_{act} : f_n = f_{n-1}$$

4.4 TEPE and AVATAR

To provide evidence for the effortless integration of TEPE into UML/SysML environments, we draw on two examples, namely the AVATAR profile and the DIPLODOCUS profile. While DIPLODOCUS has been extensively discussed in chapter 2, AVATAR is surveyed in the following to convince the reader of its different nature. AVATAR is much less tied to physical constraints, hardware software partitioning and DSE. AVATAR does not offer a separation between architecture and application concerns. The common ground of both profiles is data abstraction and the fact that algorithms are abstracted in the form of symbolic operators. The primary goal of AVATAR is to verify high level models by simulation and formal verification and to conduct schedulability analysis. Instead of renouncing the graphical support of SysML and writing complicated logical formulas, this section suggests to take advantage of the convenience of TEPE.
Recently, the profile has been enhanced to make models amenable to the verification of security properties [95], such as confidentiality, integrity, authenticity, and freshness.

4.4.1 AVATAR Methodology

The AVATAR methodology comprises the following stages:

1. **Requirement capture**. Requirements and properties are structured using AVATAR Requirement Diagrams. At this step, properties are just specified in plain text and named with a label.

2. **System analysis**. A system may be analyzed using usual UML diagrams, such as Use Case Diagrams, Interaction Overview Diagrams and Sequence Diagrams.

3. **System design**. The system is structured in terms of a Block Diagram and at most one State Machine per SysML block to describe its behavior.

4. **Property modeling**. Properties are refined and formalized within TEPE Parametric Diagrams (PDs). Since TEPE PDs involve elements defined in system design (e.g, a given integer attribute of a block), they may be constructed only after a first system design has been performed.

5. **Formal verification**. The latter can finally be conducted over the system design and for each property.

Once all properties are proved to hold, requirements, system analysis and design, as well as properties may be further refined. Thereafter, and similarly to most UML profiles for embedded systems, the AVATAR methodological stages are reiterated. Having reached a certain level of detail, refined models may not be amenable to formal verification any more. Therefore the generation of prototyping code may become the only realistic option [12].

4.4.2 AVATAR Block and State Machine Diagrams

Apart from their formal semantics, AVATAR Block and State Machine Diagrams only have a few characteristics which differ from their SysML counterpart.
An AVATAR block comprises a list of attributes, methods and signals. Signals can be transmitted on synchronous or asynchronous channels which are represented by connectors linking two ports. Connectors are associated to a list of signals.
A block constituting a data structure merely contains attributes. A block modeling a sub-behavior of the system must define an AVATAR State Machine.
AVATAR State Machine Diagrams are built upon SysML State Machines, including hierarchical states. AVATAR State Machines further enhance the SysML ones with temporal operators:

- **Delay:** $after(t_{min}, t_{max})$. It models a variable interval during which the activity of the block is suspended, waiting for a delay between t_{min} and t_{max} to expire.

- **Complexity:** *computeFor*(t_{min}, t_{max}). It models a time during which the activity of the block actively executes instructions, before transiting to the next state: that computation may consume between t_{min} and t_{max} units of time.

The combination of complexity operators (*computeFor*()), delay operators, as well as the support of hierarchical states - and the possibility to suspend an ongoing activity of a substate - endows AVATAR with features essential for real-time system schedulability analysis.

4.4.3 Harmonising AVATAR and TEPE

The correspondence between TEPE signals, attributes and blocks and the AVATAR model is straight forward as AVATAR exhibits exactly the same primitives. AVATAR expresses behavior with state machines, and therefore its concepts closely relate to labeled transitions systems.

4.4.4 System design example

The block diagram (see Figure 4.6) gives a first impression of an AVATAR block diagram. It is structured in such a way that the main block called *MicroWaveOven* represents the overall system and encapsulates the components the oven consists of. A block named *Controller* is the heart of the system and manages all peripheral equipments, that is a control panel comprising a start button and an LED indicating an ongoing heating process (block *ControlPanel*), the heating unit (block *Magnetron*), a *Bell* informing the user about the completion of the heating process, and finally a sensor element providing information about the current position of the door (*Door*). The links between blocks shown in the block diagram symbolize a synchronization mechanism based on rendez-vous. For instance, *Controller* and *ControlPanel* synchronize with each other by means of three distinct signals. Signals may be named differently in each block, so they need to be explicitly plugged to each other which is visualized with labels drawn next to the respective port.
Once the relation between TEPE signals/attributes and the underlying MoC of the system model has been established, TEPE properties are always formulated in the same way. That is why the properties studied in section 4.5.2 would apply equally well to the AVATAR model presented here.

4.5 TEPE and DIPLODOCUS

In section 4.1 it was argued that the only assumption TEPE makes on the system model is that it discriminates events and attributes. For Models of Computation stemming from labeled transition systems, the association is straight forward. We will demonstrate that with the example of the DIPLODOCUS profile. TEPE attributes can be directly related

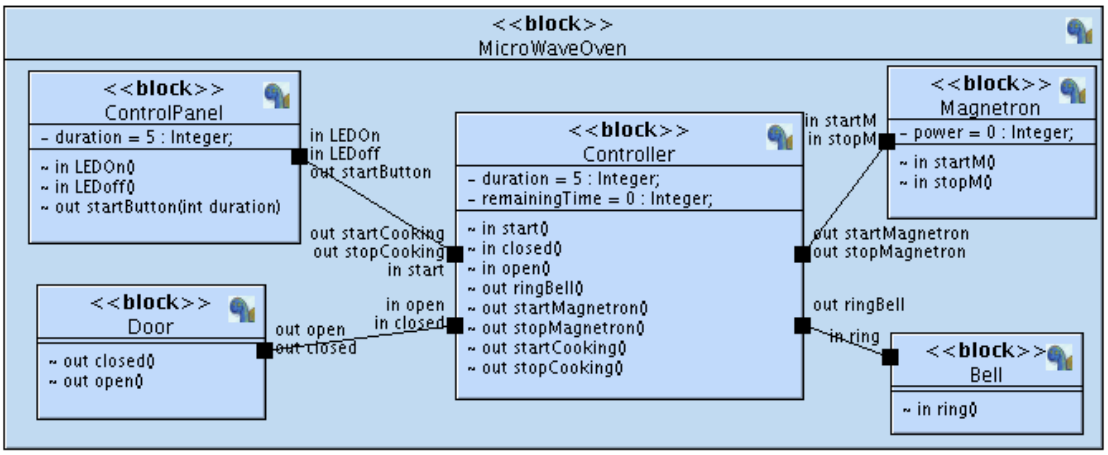


Figure 4.6: Microwave oven case study: Block Diagram

to task variables in DIPLODOCUS, and the scoping established by DIPLODOCUS tasks is expressed with Blocks. However, the correspondence between TEPE signals and operations in a DIPLODOCUS model deserves further attention. Moreover, this section addresses implementation issues of the TEPE constraint verifier which was integrated into the simulation engine (cf. chapter 5).

4.5.1 Harmonising DIPLODOCUS and TEPE

In DIPLODOCUS, tasks may be executed concurrently on different hardware components. Due to that concurrency semantics, events[1], denoting for instance the beginning and the end of transactions, may happen to occur at the same time instant as well. TEPE Logical and Sequence constraints however require signals to be totally ordered. The following three approaches may remedy that problem:

1. Imposing an arbitrary order satisfying particular criteria, for instance that the simultaneous occurrence of signals does not make a property (resulting from Logical and Sequence constraints) fail. In case simultaneity of signals is considered as an incorrect system behavior, TEPE time constraints could be used to detect it.

2. Introducing a third valuation of a property besides *true* and *false*, indicating that the property cannot be evaluated (for instance "undefined")

3. Considering all possible interleavings of signals as distinct paths and relying on TEPE path operators to determine the resulting property

So far, we opted for the first solution just to simplify the implementation of the TEPE verifier which will be surveyed in section 4.5.3. However, future experiences and insights to be gained with case studies may bring us to reconsider this decision.

[1] as defined in 3.2, not to be confused with DIPLODOCUS events

Another issue is how operations in DIPLODOCUS models should translate into TEPE signals. Obviously, one identifies application and architecture components as potential sources of signals: Tasks, commands, channels, events and requests belong to the application domain and CPUs, buses, memories, bridges and hardware accelerators to the architecture domain. All these components are subsumed in the **Source Type** category. **Source ID** specifies the name or ID of the concerned DIPLODOCUS element. Transactions are defined as an executable portion of a command (cf. section 2.3.4) and play a pivotal role for the execution semantics of DIPLODOCUS (see section 5.4). For this reason, a TEPE signal may be associated with the *start* or the *end* of a transaction. These two options are captured by the **Synchronization** option. Lastly, signals can be classified according to the type of transaction they are associated with. These **Transaction Types** are named in the same way as the DIPLODOCUS operators that issue the transaction: *Send*, *Receive*, *Read*, *Write*, *Exec*. In summary, a combination of **Source Type**, **Source ID**, **Synchronization** and **Transaction Type** accurately identifies an event (again in the MoC sense) in a DIPLODOCUS model. In the context of DIPLODOCUS, the notion of signal thus corresponds to a possible valuation of the aforementioned categories labeled with a unique identifier. The next section exemplifies the mapping of DIPLODOCUS events to TEPE signals.

4.5.2 Example

To illustrate the compatibility of TEPE and DIPLODOCUS, we get back to the example of a microwave oven, introduced in section 4.4.4. Table 4.1 demonstrates how the TEPE signals *open*, *closed*, *ringBell* referred to in Figure 4.7 relate to DIPLODOCUS elements. Therefore, we rely on the above mentioned categories. The first three TEPE signals originate from DIPLODOCUS *send* transactions of the event indicated in the *Src ID* column. The TEPE signals are thereby synchronized to the end of the send transaction. The cooking process of the microwave oven is assumed to be modeled with an Execi command, carrying the ID 221. The *startCooking* TEPE signal announces the beginning of the associated DIPLODOCUS transaction, and the *stopCooking* signal its termination. Note the respective settings of the *Synchronization* option. The explicit assignment of TEPE attributes to DIPLODOCUS task variables has been omitted, as it is straight forward. TEPE blocks are assumed to carry the same name as DIPLODOCUS tasks, just as TEPE attributes and DIPLODOCUS task variables.

4.5.2.1 Requirements

A model of a microwave now serves as an example to illustrate the ease of use and the expressiveness of TEPE. Four functional safety-related requirements have been identified and modeled in a Requirement Diagram:

- **Req1**: The heating unit is not started if the door is open.
- **Req2**: When the bell rings, the cooking time must be expired.

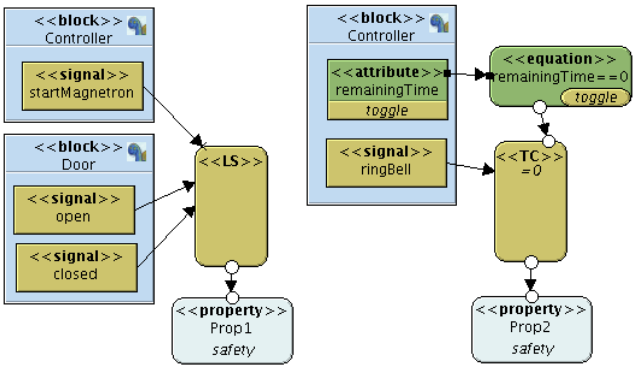
Src Type	Src ID	Sync	Trans Type	TEPE ID
Event	door_open	End	Send	open
Event	door_closed	End	Send	closed
Event	ringBell	End	Send	ringBell
Command	ID221	Start	Exec	startCooking
Command	ID221	End	Exec	stopCooking

Table 4.1: Microwave Oven: TEPE signals and DIPLODOCUS elements

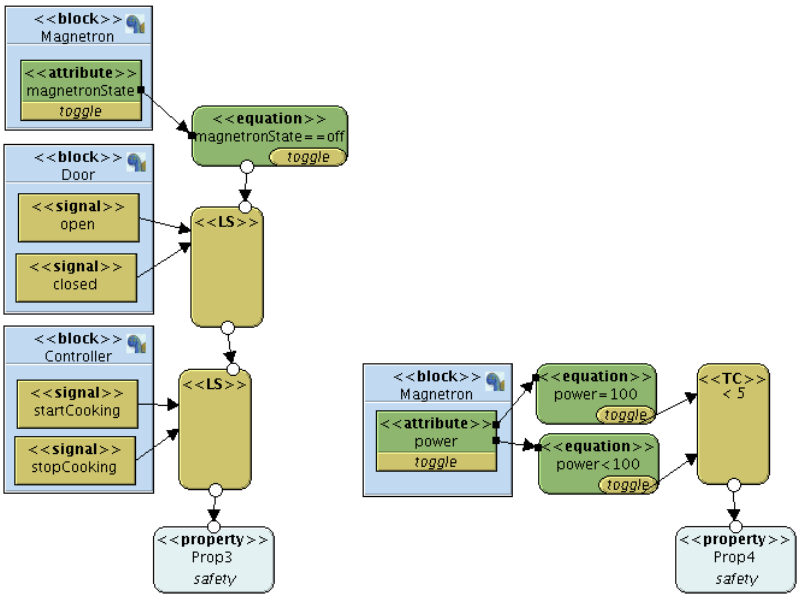
- **Req3**: When the door is opened during operation, the magnetron is switched off for the time the door remains open.
- **Req4**: To avoid an overload of the magnetron, it should not be operated for more than 5 consecutive time units at full power.

4.5.2.2 Property modeling

After having assigned a semantics to TEPE blocks, attributes and signals in a DIPLODOCUS context, the developer may proceed with the formal model of the properties to be verified, corresponding to requirements (*Req1* to *Req4*). As we will show in this section, the requirements naturally translate into TEPE.
Req1 (depicted in Figure 4.7a) is expressed in terms of a Sequence Constraint which is applied to three input signals: between occurrences of the *open* and *close* signal sent by the block *Door*, the occurrence of the *startMagnetron* signal originating from the *Controller* is considered as an abnormal system behavior. Therefore, the latter is marked as negated by means of a small cross.
The second requirement *Req2* (see Figure 4.7b) demonstrates the usage of the *Temporal Constraint* and shows its ability to express simultaneity. The remaining cooking time, stored in the variable *remainingTime* of block *Controller*, should evaluate to zero at the time instant when the bell rings.
Req3 (see Figure 4.7c) makes use of the TEPE feature to vertically compose constraints in order to interleave the respective intervals. The lowermost constraint characterizes the time interval when the oven is in operation by referring to the signals *startCooking* and *stopCooking*, both raised by the *Controller*. The second *Logical Sequence Constraint* examines the occurrence of the signals *open* and *close* during the aforementioned interval and makes sure that the magnetron is switched off while the door remains open.
Finally, it should be reemphasized that the evaluation of equations may trigger signals as well. In *Req4* (shown in Figure 4.7d) for instance, the first *Equation Constraint* studies for a value change of the power attribute of block *Magnetron* from $power \neq 100$ to $power = 100$. The second *Equation Constraint* watches for the opposite value change. The time elapsing while the magnetron is operated at full power is that way constrained to be smaller than 5 time units.



(a) Property 1 (b) Property 2



(c) Property 3 (d) Property 4

Figure 4.7: Microwave oven case study: Properties

4.5.3 Implementation Issues

This section covers the verification engine for TEPE properties, which was integrated into the overall simulation environment for DIPLODOCUS models. However, the engine was designed with the objectives to (1) keep it independent of the model to be verified, (2) architecturally separate state and event formalisms and to (3) easily incorporate potential new operators, as TEPE is still in its infancy. In this context, the simulation environment is abstracted to a mere generator of signals and attribute values. A detailed discussion of the simulation environment is deferred to chapter 5. A dedicated layer between the simulator and the TEPE verifier converts representations of signals and attributes. In so doing, the mutual independence of simulator and verifier is achieved.

4.5.3.1 TEPE Verifier Architecture

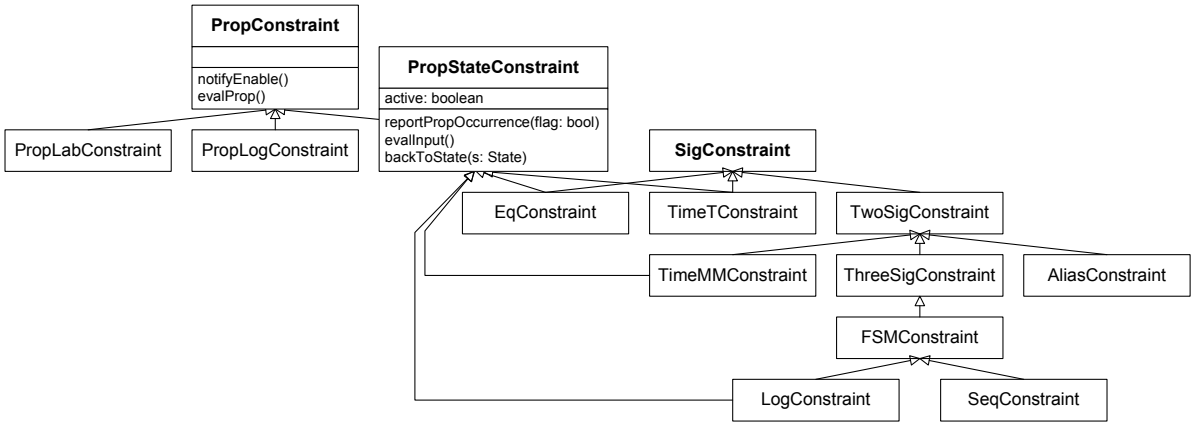


Figure 4.8: Architecture of TEPE constraints

Two separate base classes account for the horizontal and vertical semantics of TEPE constraints: *SigConstraint* and *PropConstraint*. *PropConstraint* defines an interface for controlling the activation period (*notifyEnable* method) and for evaluating the property output (*evalProp* method) of constraints. *PropLabConstraint* and *PropLogConstraint* directly inherit from *PropConstraint* and implement Property Logic and Property Label constraints.
The *PropStateConstraint* class is endowed with a flag indicating whether a constraint is currently activated and implements the method *evalProp* and *reportPropOccurrence*. Both methods are discussed in section 4.5.3.2 in more detail. *reportPropOccurrence* is invoked with a boolean parameter whenever a logical, sequence or time constraint was satisfied or violated. That way it keeps the property value of the constraint up to date and realizes TEPE's default path quantifier (in general). *PropStateConstraint* serves as base class for all constraints treating signals (except for *AliasConstraint*) and defines the *evalInput* method. The latter is implemented by all constraints receiving signals and executed

whenever a signal at the input of a constraint toggles. Section 4.5.3.3 elaborates on how the Sequence constraint implements this method.

TEPE constraints that observe the occurrence of signals inherit from *SigConstraint*. This interface provides the basic functionality for one signal input and one signal output. The inheritance structure of constraints is governed by the number of signals they may be connected to, and whether they can be represented in terms of an untimed finite state machine. It might seem astonishing that an *EqConstraint* (short for Equation constraint) inherits from *SigConstraint* for it is not supposed to evaluate any TEPE signal. However, for technical reasons, the change of equation results is broadcasted by means of signals. *TimeTConstraint* realizes semantics no. 4 of the Time constraint (involving only one time value *T*). *TwoSigConstraint* is the interface for all constraints evaluating more than one signal. *AliasConstraint* and *TimeMMConstraint* (implementing semantics 1-3 of the Time constraint, involving T_{min} and T_{max}) are direct subclasses of *TwoSigConstraint*. *ThreeSigConstraint* enhances the latter with a third signal and constitutes the base class of *FSMConstraint*. The latter subsumes constraints monitoring sequential behavior, independently from physical time, namely Logical (*LogConstraint*) and Sequence (*SeqConstraint*) constraints.

4.5.3.2 Tree and Path Quantifiers

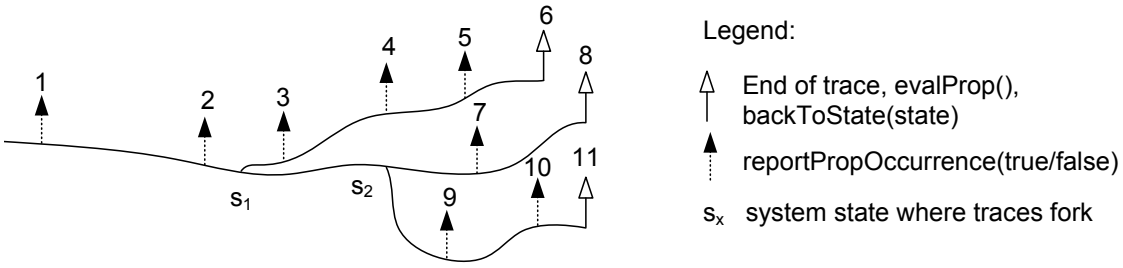


Figure 4.9: Example reachability graph with invoked verifier methods

This section elaborates on how violations and correct occurrences of signal sequences are logically combined to obtain the resulting property of one possible execution. In turn, results of several branches are logically combined to derive the final property of the whole tree of possible executions. Figure 4.9 puts the invocation of *evalProp*, *reportPropOccurrence* and *backToState* methods in the context of an abstracted reachability graph. Table 4.2 examines the response of the respective methods (columns), for Time, Sequence and Logical constraints in row one and for Label constraints in row two. To simplify matters, the constraints subsumed by row one are henceforth called R1 constraints. The variable *prop* extensively used in table 4.2 refers to the property result of a

constraint. The example reachability graph embraces three possible system executions, originating from the two states s_1 and s_2 where traces fork. All along a trace, a constraint can be satisfied (p=true) or violated (p=false), which results in a call to *reportPropOccurrence* with parameter p. In Figure 4.9, this is symbolized by arrows with a black head. Table 4.2 reveals that the parameters passed to *reportPropOccurrence* are simply logically combined with the current property value to obtain its new value. For the default TEPE path quantifier (*in general*) a logical *and* is applied. The *eventually* semantics could be realized with a logical *or*.
At the end of a trace (arrows 6, 8, 11 in Figure 4.9), the simulator sends an *evalProp* message to the lowermost of all vertically chained constraints. R1 constraints return the conjunction of their own property value and the one of the constraint above. That way, the final property of a set of vertically chained constraints is the logical conjunction of all constraints in the set. At the end of a trace, exhaustive simulation (see chapter 6) demands for the recovery of a past system state to explore another execution. For instance, at the end of trace one (arrow 6 in Figure 4.9), the simulator might get back to state $s1$ to explore the path towards arrow 8. A state change of simulation to $s1$ must be accompanied by a state change of R1 constraints, which need to recover the past value of their property (variable *prop*) at $s1$. This is symbolized by the invocation of *propertyAt(s)* in *backToState(s)*, where in this particular case s would be equal to $s1$. Label Constraints (row two in Table 4.2) act as tree quantifiers and therefore update their property output only at the end of a trace (arrows 6, 8, 11). Hence, Label Constraints do not need to implement the *reportPropOccurrence* method. The realization of Liveness and Reachability quantifiers within *evalProp* of Label Constraints resembles the one of path quantifiers of R1 constraints in *reportPropOccurrence*. However, Label Constraints combine the current value of *prop* with the property value of the constraint directly above. Label constraints are not reset at the end of a trace so as to keep track of all executions of the system. The *backToState* method is therefore left empty.

4.5.3.3 TEPE Constraints

The particular behavior of Sequence, Logical and Temporal constraints mainly stems from a customized implementation of the *evalInput* method defined in the *PropertyStateConstraint* class. Even if this section draws on the example of the Sequence Constraint (*SeqConstraint* class), it illustrates the common ground of the implementation of *EqConstraint*, *LogConstraint*, *TimeMMConstraint*, and *TimeTConstraint*. The behavior description in terms of a state machine and its synchronization to input signals is common to all these constraints. Figure 4.10 structures the behavior of constraints into two concepts: a petri net with 4 initial places, an intermediate place containing a state machine and 2 output places. The notation suggests that transitions of the state machine are triggered by the availability of tokens in all input places representing fluents f_{i1}, f_{i2}, f_{neg} and f_{n-1}. In turn, each transition of the state machine produces 2 output tokens, namely f_n and f_{out}.
In accordance with the definition of fluents in section 4.2.2, a token specifies the value of

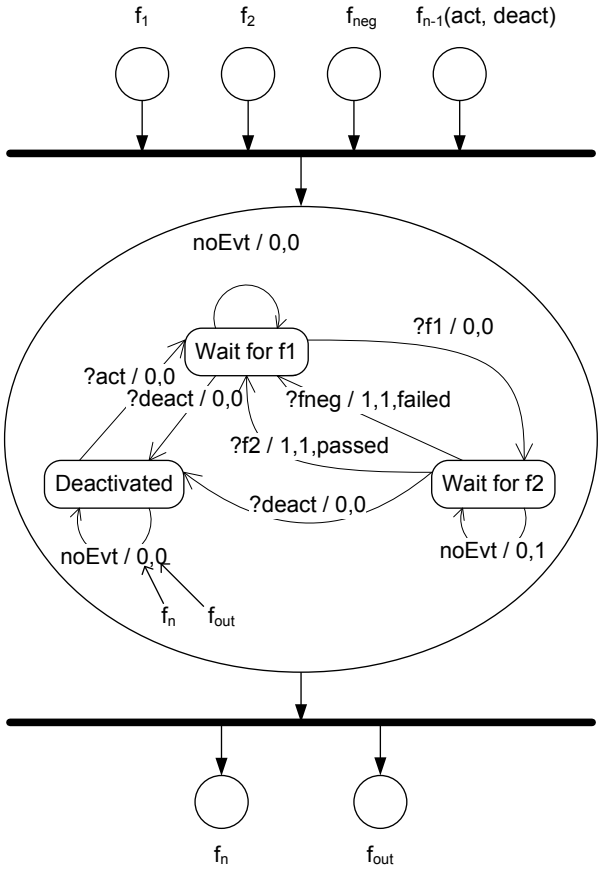


Figure 4.10: Functional view of the Sequence constraint

op \ method	**reportPropOcc(flag)**	**evalProp()**	**backToState(s)**
Constraint	*general*: prop &= flag *eventually*: prop | = flag	*return* prop & above.evalProp()	prop = propertyAt(s)
Label		*Liveness*: prop & = above.evalProp() *Reachability*: prop | = above.evalProp() *return* prop	

Table 4.2: Implementation of Tree and Path quantifiers

a fluent (0 or 1) at a particular simulation time instant. A transition in the state machine is taken if the value of the corresponding fluent is equal to 1. The notation of transitions comprises the fluent serving as trigger and right to the slash the output values of f_n and f_{out} in that order. The complement *passed/failed* implies an invocation of *reportPropOccurrence* with parameter *true/false*. To simplify the state machine, two additional triggers have been introduced: an *act* transition is taken if the previous value of the f_{n-1} fluent was 0 and the current is 1. *deact* denotes the inverse edge, thus a value change of f_{n-1} from 1 to 0. Obviously, *act* and *deact* can never occur at the same time instant. However, it is possible that more than one token carries the value 1. In that case, a total order of signals is imposed as described in section 4.5 and one transition (if available) per *1*-token is taken. If all fluent values are zero, the transition *noEvt* is taken by default. If several transitions are executed at the same time instant, only the last value of the output fluents is taken into account.
The state machine itself is trivial and just makes sure that the semantics of the Sequence Constraint is respected, which has been elaborated in section 4.3.6.3.

4.6 Conclusion

The TEmporal Property Expression language, or TEPE for short, customizes SysML parametric diagrams. Properties are built upon logical and temporal relations between block attributes and signals. TEPE diagrams are structured in a two dimensional way, where system states are related vertically and transitions between these states, called signals, are propagated horizontally. To prevent that errors in reasoning transfer from design to verification stage, TEPE enforces a formalism different from state charts and sequence diagrams. While being formally defined, the methodology may appeal to

designers less familiar with temporal logic. The comparison with FLTL revealed that especially the composition of TEPE constraints translates into sophisticated, lengthy formulas. Although nothing prevents from using the textual form of TEPE, the graphical representation based on Parametric Diagrams outreaches the latter in terms of readability. Moreover, an adequate coloring of operators facilitates the clear distinction between timed (signals) and untimed parts (properties) of the diagram.
As an OMG-SysML compliant language, TEPE may seamlessly be integrated in a broad variety of SysML real-time profiles defining notions similar to signals and attributes. The granularity and the abstraction level of diagrams is in line with high level system models.This has been demonstrated by means of the two representative UML/SysML profiles AVATAR and DIPLODOCUS. TEPE makes the underlying formalism transparent to the user while providing an expressiveness comparable to temporal logic.
TEPE opens the door for an automatic verification on the fly during simulation. Some insights into implementation issues provided evidence for the feasibility of an automated verification of TEPE properties. The verifier discussed in this section is interfaced to a DIPLODOCUS simulator through abstraction layers to keep simulation and verification matters separated. The next chapter examines the simulation part of the DIPLODOCUS environment.

Chapter 5

An efficient Simulation Engine

5.1 Introduction

The previous chapter has advocated a formal and graphical language to express system properties. To pave the way for verification of these properties, we first have to breathe live into UML models by making them executable. While the transformation of UML models into executable code is demonstrated in chapter 7, this chapter suggests a novel simulation approach for high level models of Systems-on-Chip. In the introductory part, we motivate our decision not to rely on the widely-used SystemC simulation library (Section 5.3) and review the Discrete Event MoC from a simulation performance perspective in Section 5.2. Section 5.4 reveals the simulation semantics of DIPLODOCUS and covers assumptions made on architecture, application and mapping. To get a bit more concrete, abstractions with respect to the separation of concerns are illustrated with the example of the field bus standard CAN in Section 5.4.4. Section 5.5 elaborates on the simulation strategy and makes the reader familiar with simulation phases and synchronization mechanisms. For details about some intricate implementation issues, the reader may refer to Section 5.6.
Our fast simulation approach is centered around two basic principles:

- A modeling methodology which abstracts both data and functionality.
- A simulation strategy which exploits efficiently the properties of high level models. Thereby, the granularity of the simulation matches the granularity of the application model.

This implies that the simulation speed highly depends on the way the application is modeled. Indeed, the more coarse grained the model, the longer the contiguous actions considered by the simulator (the so called transactions) and the more efficient (in terms of cycles/sec) is the simulation. If synchronization of concurrent processes does not require transactions to be truncated, the simulator executes transactions spanning many clock cycles as a whole.

5.2 Discrete Event MoC revisited

A general introduction to the Discrete Event (DE) MoC was given in Section 3.2.3. This chapter provides a more practical view on the topic and places emphasis on simulation and implementation aspects.
The focus of DE simulation may be oriented towards two building blocks of a MoC [30; 37]: events and processes (cf. Section 3.2). Three perspectives also referred to as world views are prevalent in DE simulation. The "event scheduling" and the "activity scanning" view are event centric, whereas the "process interaction" view focuses on processes, as its name suggests.

In the **event scheduling view**, systems are expressed in terms of events to which a particular handler, e.g. a behavior is attached. The handler embraces event specific actions to be taken (calculation, modification of system states) as well as notification and cancellation of events. Consequently, the simulation procedure consists in selecting the earliest event, advancing simulation time accordingly and carrying out the corresponding event handler.
The **activity scanning view** is similar to the event scheduling view but enhances the latter with contingent events. The term refers to events, whose occurrences are not triggered by time but by a condition, being a function of the system state. A contingent event usually denotes the beginning of some activity and triggers another event that stands for the end of that activity. In addition to the above mentioned simulation procedure, all conditions of contingent events have to be evaluated upon simulation progress. This significantly slows down the simulation.
In contrast to the previous world views, **process interaction** defines a system in terms of actors implicitly triggering events. Processes capture sequences of actions and event handling primitives, potentially augmented with control flow operators. Dedicated instructions may suspend the execution of a process for a certain amount of time, or until a resource is available. That way, the process traverses the states "suspended", meaning that it is waiting for time to elapse or condition to be met and "active" if the process is scheduled for reactivation at a definite time instant.
Each simulation round is thus structured in three phases: first, simulation time is advanced to the earliest activation time of the active processes. Second, processes with the earliest activation time are executed until they suspend. Third, conditions of idle processes are checked and the processes are reactivated if necessary. The second and the third step are iterated until a steady state is reached where no processes are activated. After that, a new simulation round is entered.
The process interaction view has been adopted by various RTL simulators for VHDL, Verilog and also SystemC. The next section elaborates on the execution semantics of SystemC and to what extend it could be reused to simulate DIPLODOCUS models.

5.3 SystemC - Virtues and Vices

SystemC is an extension to the C++ language introducing a notion of concurrency and time. It is based on an event-driven simulation kernel and very capable when it comes to jointly modeling hardware and software at various levels of abstraction. It is debatable whether SystemC can be seen as a language on its own or simply as a C++ library embracing classes, macros and datatypes. The designer is provided with facilities that mimic a concurrent real time environment, such as concurrent processes, delayed and resolved signals, structural hierarchy, connectivity and clock cycle accuracy.

As the semantics of clock cycle accurate SystemC (as defined in [85]) bears resemblance with VHDL and Verilog, SystemC also features the process interaction view of the Discrete Event MoC (cf. Section 5.2). Concurrency should be expressed in terms of distinct processes, in order to fully leverage the functionality of the kernel. In the case of DIPLODOCUS, that means that every abstract task should have a corresponding counterpart in SystemC. If we were to represent the semantics of the application model alone, this would be an efficient way of doing. However, as a DIPLODOCUS simulator is faced with constraints imposed by the architecture, the aforementioned methods seams much less appealing. As the level of concurrency decreases significantly when considering shared resources, the usage of more SystemC tasks than there are active hardware components would be wasteful. Work concerned with SystemC simulation performance [87; 97; 123] reveals that the open source simulation kernel leaves considerable room for improvements. The overhead of task maintenance is mostly due to an immediate execution of SystemC tasks. As opposed to our methodology, there exists no model transformation besides the compilation and binding with the SystemC library. This entails that every context switch that hands over control to the simulation kernel must guarantee the proper setting of registers. For that reason, the performance of context switches suffers from the necessary save and restore operations. An explicit transformation stage allows us to decompose tasks into sections between potential preemption points and to execute them in the context of a task object. Context switching is thus reduced to passing a reference to a respective task data structure and executing an atomic section with respect to that reference. In conclusion, the mapping of tasks on concurrent hardware entities should avoid the SystemC task primitives and rather be built on an individual solution, tailored to the DIPLODOCUS semantics.

A SystemC simulation round is structured in accordance with the basic requirements of DE simulation and the three supported task variants. Tasks are classified into methods, threads and clocked threads. Clocked threads are evaluated upon an advancement of simulation time, whereas the signals to which methods and threads are sensitive have to be defined. Methods and threads differ in the sense that in threads control has to be relinquished explicitly (e.g. wait command), whereas the last statement of a method is implicitly interpreted as a preemption point. The complexity of the kernel originates from the sophisticated semantics of tasks and their potentially complex activation con-

ditions. When all tasks are suspended, the kernel first generates events for all active clocks. Thereafter, methods and threads are evaluated until the last statement or a wait statement is encountered. After clocked threads have been scheduled for execution, all modified signals must be updated. This can cause other events to be notified, and in turn tasks may have to be executed once again. As soon as this loop terminates, i.e. no further events are generated, clocked threads are evaluated, simulation time is advanced and finally clocks are updated.
Thanks to the simple but expressive semantics of DIPLODOCUS, its simulation does not demand for the support of such involved task and signal types. Even DIPLODOCUS tasks receiving requests can be represented with a simple preemptive task model. In the DIPLODOCUS MoC, communication is not immediate and is defined to consume simulation time, thus preventing zero delay feedback loops. For this reason, tasks only need to be evaluated at most once per simulation round. Moreover, synchronization is accomplished in an asynchronous fashion and so no logic is needed to resolve the value of shared signals, like in SystemC.
In summary, as (1) we want to stress the SystemC concurrency model the less possible and (2) extensive parts of the kernel are irrelevant for DIPLODOCUS models, we would end up reusing mainly the SystemC DE engine. As it is explained in 5.5, even this part can be optimized with the DIPLODOCUS semantics in mind. This is where our decision to refrain from using SystemC comes from.

5.4 DIPLODOCUS' Simulation Semantics

In the subsequent three sections, the simulation semantics is informally presented and classified into the three categories: Application, Architecture and Mapping. Assumptions were made for three reasons: first, some of them stem from abstractions inherent to the DIPLODOCUS model of computation which is especially tailored to performance aspects. In the following, these assumptions are marked with (Perf). Other assumptions (denoted by (Sim)) are either technical and simply facilitated the implementation of the simulator or are of descriptive nature. A relaxation of these assumptions is envisaged in the future. Assumptions indicated by (Sys) are normally driven by particularities of the system to be modeled. (Sys) assumptions were obtained from the insights gained in various projects and case studies.

5.4.1 Application

Concerning the application model, the following assumptions were made:

- (Perf): A transaction is considered to be monolithic in the sense that the partial order of actions within a transaction is not resolved. Branch prediction penalties are spread uniformly across a transaction (cf. section 5.4.2), and so it is not intended to specify their start and finish time. This assumption comes with the positive effect

of reducing indeterminism and augmenting simulation performance while being justifiable at the given level of abstraction. Performance figures primarily depend on which control flow path in a DIPLODOCUS task is taken. The respective guards (of Choice commands) are mostly time invariant, as long as they do not refer to a value obtained from a Notified Command. (Recall: This command returns the number of events stored in a FIFO). By assuming a scarce use of Notified Commands, the overall workload imposed on the hardware architecture is largely independent of the timing. Therefore, given the inherent inaccuracy of high level models, the partial order within transactions is assumed to play a marginal role for performance measurements.

- (Perf): As a rule of thumb, control flow related operators do not advance simulation time whereas commands triggering transactions indeed do. The following commands are executed in zero time: Action, Choice, For Loops, Sequence, Random Sequence, Random Number. Other commands let time elapse: Write Channel, Send Event, Send Request, Read Channel, Wait Event, Notified Event, Select Event, ExecI, ExecC, Delay. The assumption requires that computational complexity and data transfers are concentrated in Write, Read and Exec commands. However, this is not a limitation as the latter operators can be placed anywhere in the task.

- (Sys): In the simulation framework, synchronization is expected to have an impact on system performance. The cost of Send Event/Wait Event commands is provided as a parameter to the simulator. In case events are mapped onto a bus the cost is expressed in terms of bytes, otherwise it is considered as an ExecI unit. However, in practice it can often be argued that the performance impact of data transfers due to synchronization is neglectable with respect to the rest of the application. This heavily depends on the average length of computations and data transfers of the system to be modeled. The simulator is flexible enough to be adjusted to the different circumstances.

- (Sim): The application model embraces indeterministic commands such as Random, Random Sequence, Random Choice, ExecIInterval, ExecCInterval, DelayInterval. During simulation, indeterminism is simply resolved by means of a random number generator, whereas formal techniques and the exhaustive simulation explore all valuations of the particular random variable. This complies to prevalent model checking practices.

- (Sim): So far, Select Event commands are defined to be deterministic. Events are checked in the order in which they were connected in the graphical model.

5.4.2 Architecture

In this thesis, the impact of caches, memories, CPU power saving strategies and operating systems are modeled in a rudimentary way. To increase the accuracy of simulation

results, the proposed models should be refined and calibrated according to the used cache hierarchy, memory technology, power manager and operating system. In the field of simulation, the purpose of this work is to propose an efficient simulation strategy tailored to the properties of DIPLODOCUS models. The simulator has been designed with future enhancements in mind, which means that it could accommodate a more sophisticated cache or energy manager model. The issue of adequately representing data and instruction caches has been investigated in the scope of a dissertation [55] simultaneously to this work. Moreover, there is an ongoing dissertation which addresses the refinement of the existing energy consumption models and their integration into the simulator.
Three deterministic penalties may be imposed on Exec operations:

- **Power saving mode**: If a CPU is not loaded with instructions for a given time, it enters an idle mode which reduces the energy consumption. When the CPU is in idle-mode, tasks requesting CPU processing time suffer from a constant wake-up delay.

- **Context switch**: The scheduler of the operating system may also degrade performance especially if task switches occur frequently. To account for the additional payload due to scheduling algorithms and context switching, a static task switching penalty is introduced. It delays a transaction on a CPU, if the previous transaction did not belong to the same task.

- **Branch prediction**: In pipelined CPU architectures, branch predictors attempt to guess which branch of a conditional instruction will be chosen. The purpose of the branch predictor is to avoid a pipeline-flush, implying that partially executed instructions have to be discarded. To account for this effect, the execution time of an Exec unit $t_{exec} = \frac{cycles_{exec}}{f_{processor}}$ is multiplied with a correction factor $t_{execi,branch} = t_{exec}\left(p_{miss} \cdot s_{pipeline} + 1 - p_{miss}\right)$, where $0 \leq p_{miss} \leq 1$ denotes the branch miss probability of a conditional branching instruction and $s_{pipeline}$ the number of stages to be flushed in case of a branch miss

Moreover, the following modeling assumption have been made:

- (Sim): The master clock frequency is assumed to be a common integer multiple of the frequency of all clocked components (CPUs, buses, memories). However, this assumption could easily be eliminated.

- (Sys): When transferring data, the transmission speed is rather limited by the interconnect than by execution components. The duration of Exec transactions is therefore calculated as a function of CPU parameters, whereas the duration of data transfers solely depends on interconnect parameters. This assumption should hold for most of the modern chip architectures. Otherwise, it could be reconsidered by slightly adapting the simulator.

- (Sim): The throughput is determined by the weakest link, meaning the slowest bus or memory, and proportional to the transmitted data (for the time being, no static offsets are added for memory accesses, but this decision could be reconsidered with minor effort).
- (Perf): Each interconnect component on a route may delay a transaction, thus recalculate the transmission time, or add a static access time.
- (Sim): Buses may be endowed with several independent communication channels which can be used simultaneously. This extends the scope of the model to other interconnect architectures like crossbars enabling multiple simultaneous connections.
- (Sim): The communication model bases on the circuit switching paradigm; for the time being packet switching cannot be represented. During the entire data transmissions on more than one bus, at least one communication channel on all involved buses has to be available and is reserved. This could be prevented by making bridges active components, that receive data on one bus and create a send transaction on another bus. Reservation is accomplished starting from the CPU towards the memory element in a causal fashion.
- (Sim): Deadlocks are not resolved in case two CPUs mutually try to get access to a bus that has already been reserved by the other one.
- (Sim): A memory has as many ports as it has connections to buses. In case a bus is equipped with several channels, all channels are assumed to have a separate memory port.
- (Perf): Hardware accelerators are modeled as CPUs with all penalties disabled. One CPU should be foreseen per task to avoid the implications of a scheduling policy. Indeed, an operation in DIPLODOCUS is only characterized by its complexity and therefore there is no fundamental difference in whether the operation is executed on a CPU or on a dedicated hardware component.
- (Sim): DMAs are represented with dedicated CPUs running tasks which accomplish the data transfer.

5.4.3 Mapping

Concerning the mapping, the following assumptions were made:

- (Sys): The amount of data carried by events may or may not be neglectable with respect to data transfers expressed with channels. Therefore, the simulation environment leaves the decision to the user whether events are mapped onto buses.

- (Sim): Data associated to an abstract channel is assumed to be located at one single physical position in the system. For that reason, a channel is implicitly mapped onto n buses, $n-1$ bridges and 0 or 1 memory. This assumption is debatable given that embedded systems nowadays comprise a heterogeneous memory architecture (volatile and non-volatile memory of different techniques). This assumption is relaxed in [55].

- (Sim): If a channel is mapped onto a memory, the route connecting sending CPU, memory and receiving CPU may be inferred by the code generator. The channel does not have to be explicitly mapped onto buses or bridges. In case there exist several routes, the intended route should be marked with at least one mapping artifact. Otherwise the result is undefined as it depends on implementation internals of the code generator.

- (Sim): If a channel is mapped onto buses, read and write operations normally involve transactions on those buses. However, if a channel is not mapped onto any memory, it means that the data is buffered somewhere within the bus master of the receiver. In this case, a read transaction is conveyed on the bus whereas the write transaction does.

5.4.4 Abstraction example: CAN bus

The decomposition of a communication standard into its implications on application, architecture and mapping model is now exemplified by means of the CAN bus. In the context of the EVITA [104] project, a vehicular on board network had to be modeled in DIPLODOCUS with the objective to obtain early performance figures. Even if new standards for field buses such as FlexRay are emerging, the CAN bus still enjoys great acceptance from manufacturers of cars, medical and industrial equipment. CAN, which stands for Controller Area Network, is a serial field bus protocol especially suited for in-vehicular use. The automobile industry witnessed the advent of an increasing amount of electronic control systems. These systems present different requirements in terms of communication data and reliability. To avoid dedicated wires for the various control applications, the need arose to reconcile the requirements which culminated in the CAN standard.
CAN is a serial bus comprising two wires. Arbitration is organized in decentralized fashion according to the CSMA/CA policy: due to dominant and recessive potential differences of the wires, a message that is transmitted with highest priority automatically overrides others. Nodes transmitting a lower priority message sense this by comparing the written potential with the actual potential on the wire, and upon unsuccessful transmission nodes back off and wait.
The documentation of the CAN standard gives various physical details relevant for pin and cycle accurate models. However three semantical characteristics have been identified which are significant at the abstraction level of DIPLODOCUS:

- Bus arbitration is based on properties assigned to messages, not to nodes.
- Messages are broadcasted to all listening nodes in the same LAN.
- A message is only transmitted once on the bus. There are no distinct transactions for a write operation, transferring the data to a memory, and a subsequent read operation, transferring the data to a controller.

At application level, DIPLODOCUS channels and events are point-to-point links by definition and thus do not support broadcasts. To achieve a broadcasting like behavior, a multi channel write operator was introduced which writes a given amount of samples to several channels. If the respective channels are mapped onto the same interconnect supporting broadcasts (like the CAN bus), data samples are only transmitted once on the bus. Otherwise, the new operator is handled in the same way as several distinct (single channel) write operators. In so doing, the separation of concerns is maintained.
At architecture level, the above mentioned twofold semantics of multi channel write operators has to be taken into account. Moreover, if the CAN protocol is selected for buses, priorities assigned to channel mapping artifacts replace priorities assigned to bus masters. That way, a conventional fixed priority scheduling algorithm can be used.
Finally, at mapping level, we have to account for the lack of a dedicated memory in a CAN message exchange. Messages are implicitly buffered in the receiving node in case they match its filter criteria. Thus, a special semantics is assigned to write operations on channels not being associated with a memory. These operations are assumed to be executed within the receiving node by reading the internal buffer, and do not require bus access.

5.5 Simulation strategy

5.5.1 Improvements with respect to conventional DES

Methods to perform conventional Discrete Event Simulation (DES) were presented in 5.2. In Section 5.3 it was argued that for DIPLODOCUS, the utilization of SystemC does not bring major benefits; just the basic DE kernel could potentially be reused. However, DIPLODOCUS simulation semantics introduced in 5.4 was conceived with simulation performance in mind. This semantics leaves room for further improvements of general DES at the expense of generality. Indeed, our simulator is as a result of this trade-off tied to high level models with a semantics similar to the one of DIPLODOCUS models. The remainder of this section addresses the process of steamlining conventional DES to the needs of high level models.
At first, the essential data structure in our simulator is not an event, but a transaction. A transaction represents a computation or communication action involving one or more hardware components. As opposed to an event, a transaction is characterized by a duration and embraces two events, denoting its beginning and its end. This policy cuts

the number of data sets to be handled to half, as compared to an event based mechanism. An intermediate objective consisted in minimizing the number of entities which may trigger an event. In our simulator, CPUs are the only active components, i.e. they are authorized to initiate transactions (recall: DMAs have to be modeled as dedicated CPUs.) Passive components belonging to the application domain (channels, etc.) and architecture domain (buses, memories, bridges) cannot generate transactions. This is due to the fact that they are supposed to work in collaboration with a CPU or a DMA. Passive components may however postpone or delay a transaction to account for bus contention, access time of a memory, etc. As the amount of active components is usually small as compared to the number of tasks, the simulator renounces an explicit event queue. This data structure normally stores scheduled events in ascending order of their time stamps allowing the simulator to simply elect the first one for execution. In our simulator, it suffices to consult the list of CPUs, query the latter for transaction proposals and select the transaction with earliest finish time. This procedure avoids the creation and destruction of events as well and other maintenance operations for event queues. In conventional DES, events may have to be canceled and removed from the queue upon unforeseen incidents like task activation or synchronization hazards. As it is demonstrated in the following section, the speculative policy of our simulator gives us the opportunity to postpone or truncate transactions just by modifying a single value.
Our DE simulation further leverages architecture semantics by hiding transactions to the main scheduler which are awaiting resource allocation. Only upon a grant of the resource, the simulation kernel has to acknowledge the respective time stamp. This is to guarantee a causal, consecutive allocation of a route consisting of several buses.

5.5.2 Basics

Unlike a simulation strategy where components schedule actions cycle by cycle, the present approach leverages the granularity of the application model. The workload (in terms of data or Exec operations) annotated to operators of the Activity Diagrams are directly seized for simulation. The layered architecture (depicted in Figure 5.1, explained in section 5.5.3) gradually reduces the concurrency from application level to mapping level. It should be reemphasized that tasks are defined to be concurrent at application level whereas at mapping level the availability of resources constrains concurrency. Each HW component is assigned a local clock which will be ahead of others as soon as the component carries out a transaction. The built-in DE simulation kernel makes sure that the transaction with earliest finishing time is served first as it could have an impact on later transactions.

As stated previously, the vital data structure assuring synchronization across several simulation layers and components is referred to as transaction. A transaction represents a computation or communication operation initiated by a task. As opposed to discrete events, transactions have a duration and may span many clock cycles, depending on the

granularity of the application model. The length of a transaction is initialized according to the application model, and more specifically it is determined by the current command of the task. At simulation runtime, a transaction may have to be truncated into smaller portions. This could be due to a data dependency (for instance a task activation due to a read/write operation) which has to be resolved by permitting the affected CPU to reschedule. Consequently, truncation of transactions is likely to happen when the tasks are heavily synchronized by means of DIPLODOCUS Channels, Events and Requests.

This chapter elaborates on the role and attributes of transactions. To account for the separation of concerns, two time scales are used to quantify the duration of a transaction at application and at mapping level. The virtual time scale refers to complexity annotations of task level operators in terms of Exec operations or the amount of data to be transferred. The virtual time scale is completely independent of architecture specific parameters (such as the speed/data rate of devices). During simulation, the absolute duration is calculated as a function of the virtual time and device parameters.
The main attributes of a transaction are listed below:

- **startTime**: Stands for the absolute point in time when a device starts executing the transaction.
- **duration**: Indicates the number of time units needed to execute the transaction. If several hardware components are involved in the execution, this value is determined in several stages
- **virtualLength**: Specifies the amount of data to read/write or the amount of processing units to carry out (example: for a READ 3 and a channel having a width of 2, the corresponding transaction spanning the whole operation would have a *virtualLength* of 6).
- **runnableTime**: Indicates the point in time when the transaction gets runnable, that is when the dedicated task is ready to execute it. As opposed to *startTime*, *runnableTime* is independent of shared resource contention.

5.5.3 Transaction passing

Figure 5.1 illustrates the way of two exemplary transactions through the simulator. The symbolic application diagram in the figure depicts two tasks comprising one exemplary command each. The transactions stem from these commands: the first transaction is a representative of a channel operation (Read/Write) and its way is marked with the suffix *a*, the second one is a symbolic execution (Execi) of an algorithm denoted by arrows with the suffix *b*. The number prefixes associated to arrows indicate the order of the different stages of simulation. The columns stand for the consecutive simulation phases, whereas the rows point out the layered architecture of the simulator.

Transactions are initially generated (arrow 1a/1b) according to the virtual length of the current operator of the task. Control flow operators which are defined not to consume simulation time (cf. Section 5.4) do not generate transactions either. During the **Evaluation Phase**, the simulator executes all operators until it encounters the first one which triggers the generation of a transaction. This evaluation of tasks based on DIPLODOCUS semantics is accomplished at a dedicated layer (called *Task Layer*).
Subsequently, during the **Scheduling phase**, abstract channels constrain the execution of Read/Write operators and the *Abstract Communication Layer* may redefine the *virtualLength* attribute (arrow 2a). If channels exhibit a finite size, write transactions may be shortened in order not to exceed the remaining space. Their *virtualLength* may even be set to zero when the channel temporarily does not allow for read/write operations. Execi transaction of course skip this stage for they do not rely on channels.
The *Execution HW Layer* is especially relevant to Exec transactions (cf. arrow 2b): the parameters annotated to CPUs such as its frequency allow to derive the real duration of the transaction in terms of physical time units. Furthermore, the scheduler of the CPU calculates the *startTime* of the next transaction to be granted CPU time. As defined in Section 5.4, channel operations are scheduled by CPUs and also are assumed to consume computing time but their duration is determined by the weakest link in the chain of interconnected buses. For this reason, merely the *startTime* of Read/Write transactions is tentatively computed (arrow 3a) although the value is reconsidered at potentially multiple cascaded *Communication HW Layers*.

The excerpt of the mapping diagram in Figure 5.1 suggests that the abstract channel is mapped onto two buses, one bridge and a memory. Buses are endowed with a scheduler which arbitrates concurrent bus accesses. Due to congestion, a transaction may experience delays, requiring its *startTime* to be postponed. Buses may further redefine the *duration* attribute in case the data transfer exceeds the current value and set the *virtualLength* to the maximum burst size (see arrow 4a) of the bus. A bus together with a slave component (such as a bridge or a memory, cf. arrow 5a, 8a) is said to constitute a *Communication HW Layer*. Slaves are not considered as active components and are therefore not able to schedule transactions. However, a memory or a bridge may introduce an additional delay, which is added to the duration parameter. In case several buses are involved in the arbitration procedure, time elapses between two consecutive grants. This requires the intervention of the Simulation kernel to guarantee causality of the simulation (depicted by arrows 6a, 7a). When a transaction has been successfully granted access to all required resources, it is finalized by the simulation kernel (**DE Phase**). The latter is in charge of ordering and truncating transactions as well as designating a transaction for execution. This lowermost layer is called DE Simulation (arrows 3b, 9a) and it is discussed in more detail in the following section.

The **Update Phase** concludes a simulation round (arrows, 10a/4b) and updates the inner state of simulation components according to the just scheduled transaction. To give an example, the variables to be updated are amongst others: the list of scheduled transac-

tions of CPUs (*Execution HW Layer*) and buses (*Communication HW Layer*) and the filling level of abstract channels (*Abstract Communication Layer*). On the *Task Layer*, it has to be determined whether the final transaction still covers the whole operator or if the latter still has outstanding transactions. If the command is finished, simulation moves on to its first successor that generates a transaction. That way, the **Evaluation Phase** is entered and the procedure starts from the beginning.

5.5.4 The simulation kernel

As prefigured in 5.5.2 the simulation kernel is located at the lowermost simulation layer and has the final say which transaction is designated for execution. From the kernel's perspective, the layers above act as a filter hiding all transactions which are not runnable either due to application constraints or due to resource contention. The *Abstract Communication Layer* filters out all transactions which are impeded by full or empty channels. Figure 5.1 is simplified with respect to the *Execution HW Layer* which is actually the only mediator between DE Simulation and the other layers. In reality, the kernel exclusively queries the *Execution HW Layer* for new transactions. In turn, this layer gathers the feedback of all involved *HW Communication Layers* before notifying its presence to the kernel.
So far, it has not been discussed in which circumstances transactions need to be truncated by the kernel. As described in Section 5.5.2, transactions arriving at the kernel may be arbitrarily long. Their maximum virtual length depends on the application model (length of commands, size of channels, etc.) and on the parameters annotated to HW resources (maximum burst size of a bus, scheduling policies, etc.). To separate model semantics from the simulation algorithm, the kernel makes no assumption on the virtual length of transactions (although in some cases this would be feasible). A problem is encountered when a CPU schedules a comparatively long transaction while another transaction, running on the same CPU, becomes runnable. This could be due to a task synchronization taking place and activating a blocked task. If the new transaction becomes runnable before the termination of the long one, the scheduler of the CPU has to be given the possibility to reschedule. This means that simulation must quit the *DE Phase*, reenter the *Scheduling Phase* which in turn calls for a truncation of the long transaction. **Simulation is speculative** in the sense that scheduling is performed with transactions complying to application and architecture semantics, leaving the resolution of synchronization conflicts to the lowermost layer. Thanks to this policy, transactions neither need to be canceled nor reinserted, and truncation becomes tantamount to merely replacing a single integer value.
The scenario depicted in Figure 5.2 illustrates the condition under which the transaction tr_2 is truncated at the finish time of transaction tr_1. We assume that transaction tr_1 was issued by command c_1 of task ta_1 which is about to be executed on cpu_1.

1. There exists a task ta_{dep} whose current command is called c_{dep}. ta_{dep} could potentially

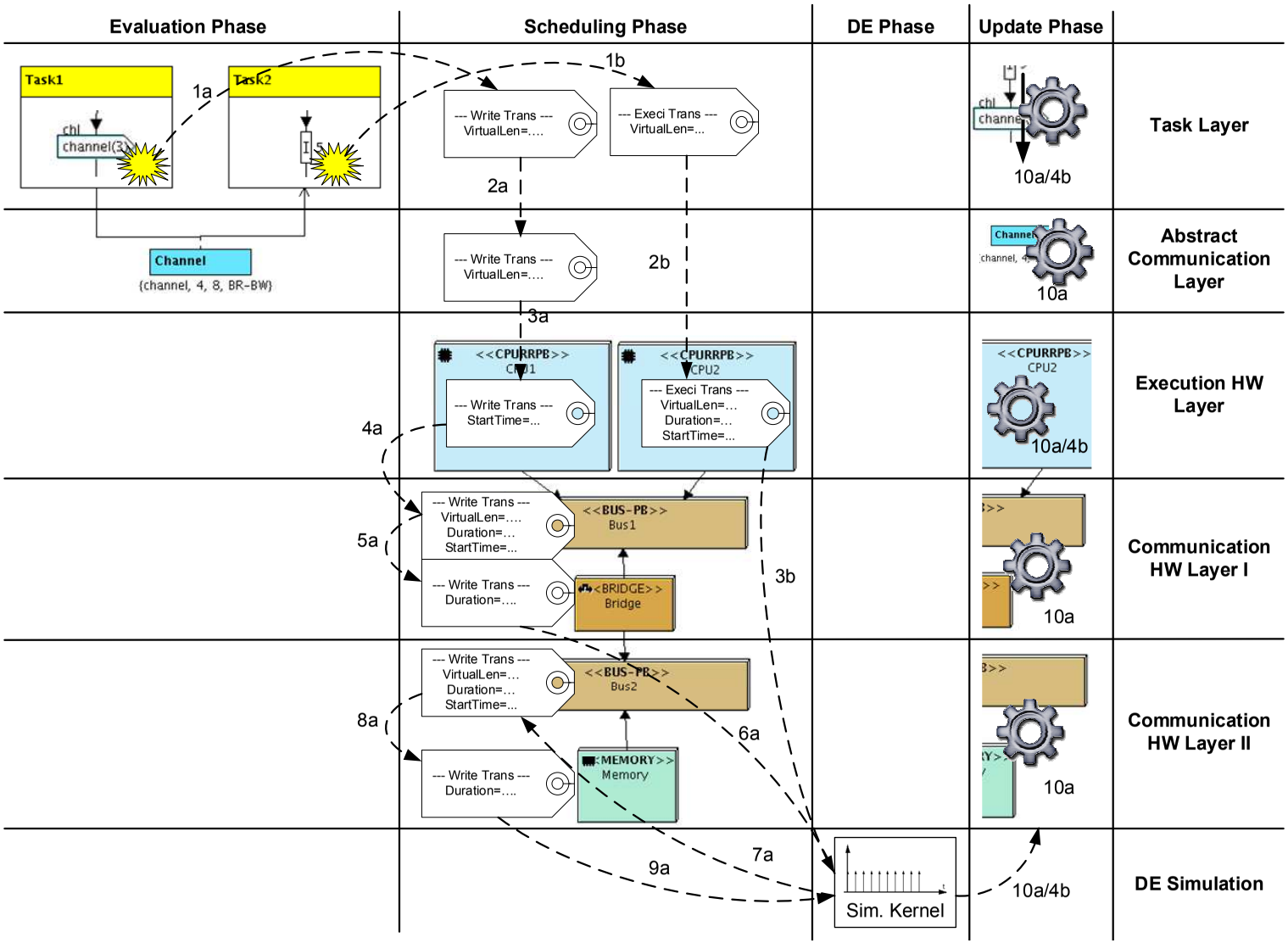


Figure 5.1: Simulation methodology at a glance

be activated by a read/write operation performed by c_1.

2. ta_{dep} is not running on cpu_1.
3. c_{dep} and c_1 access the same channel.
4. ta_{dep} proposes the transaction tr_{dep} and is therefore not blocked any more.
5. $tr_2 \neq tr_{dep}$ is scheduled on cpu_{dep}, the cpu onto which ta_{dep} is mapped.

Transactions that transfer data on a bus cannot be truncated; their size is however limited to the burst size of the respective bus.

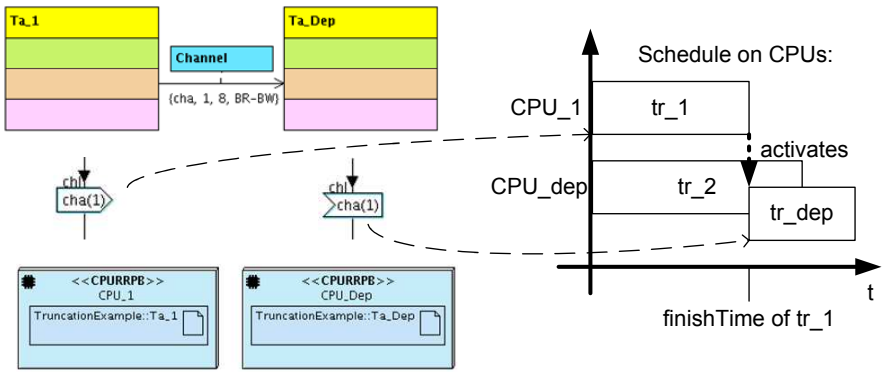


Figure 5.2: Example of transaction truncation due to synchronization

5.5.4.1 Example

To illustrate the operating mode of the kernel including truncation of transactions, we consider the three exemplary tasks depicted in Figure 5.4 together with the architecture shown in Figure 5.5. Figure 5.3 illustrates the scheduling of the transactions generated by the respective DIPLODOCUS operators. In Figure 5.4, commands are annotated with their transactions. Task1 merely comprises two Execi operators representing computations. Task2 first performs a variable assignment, which is immediate and does neither let time elapse nor trigger a transaction. The two following operators model some computations and a write operation. Task3 is supposed to read the data sample written by Task2.

The system is assumed to comprise two CPUs. Task1 and Task3 are mapped onto CPU1 and Task2 onto CPU2. As mentioned earlier, CPUs are the only active components which may signal transactions to the kernel. The layers above the kernel filter out all transactions whose execution is impeded by application or architecture constraints

(compare *Scheduling Phase*, Section 5.5.2). Both CPUs have already invoked their scheduler and in turn notify the runnable transactions $Tr1$ and $Tr2$ to them (cf. Figure 5.3a). The transaction on CPU1 is expected to finish at time $t1$, the one on CPU2 at time $t2$. As we will see, the kernel may recalculate finish times in case transactions need to be truncated. The kernel selects the transaction with the earliest finish time which is in line with traditional discrete event mechanisms. This is simply due to the policy of cause and effect: earlier transactions may defer or alter later transactions.

After the kernel has scheduled $Tr1$ as illustrated in Figure 5.3b, the *Update Phase* sets component state variables accordingly. The subsequent *Evaluation Phase* brings out the next transaction on CPU1, $Tr4$. $t2$ is the earliest finish time greater than $t1$, the current simulation time, and consequently $Tr2$ is elected for execution (Figure 5.3c). Subsequently, $Tr3$ is generated upon the next occurrence of the *Evaluation Phase*. In Figure 5.4, it appears that transaction $Tr3$ belongs to a Write operator that writes one sample to a channel. The execution of $Tr3$ makes Task3 on CPU1 runnable, as the latter is attempting to read samples from the same channel. After having scheduled $Tr3$ (Figure 5.3d), the kernel must allow CPU1 to take a new scheduling decision at time $t3$, to account for the new runnable task Task3. For that reason, the kernel cuts down $Tr4$ by $t4 - t3$ time units and subsequently initiates its execution (Figure 5.3e). The virtual length of transaction $Tr4$ is recalculated as a function of its new duration $t3 - t1$, the speed of CPU1 and the number of cycles per Execi unit of CPU1. Let us assume that the new, shortened virtual length is 3. In that case, the remaining part of the Execi operator amounts to 2 units, and consequently the virtual length of transaction $Tr6$ is set to 2. In the following *Scheduling Phase*, the scheduler of CPU1 arbitrates between the two competing transactions $Tr5$ and $Tr6$. The result depends on the respective policy of the scheduler.

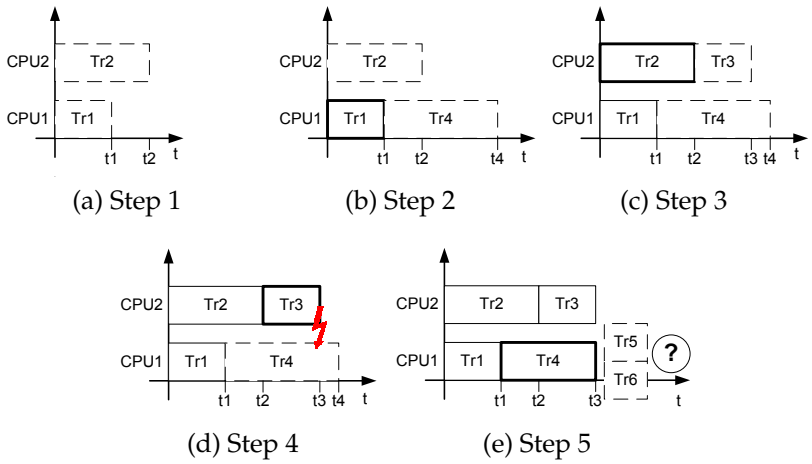


Figure 5.3: Transaction truncation inside the DE kernel

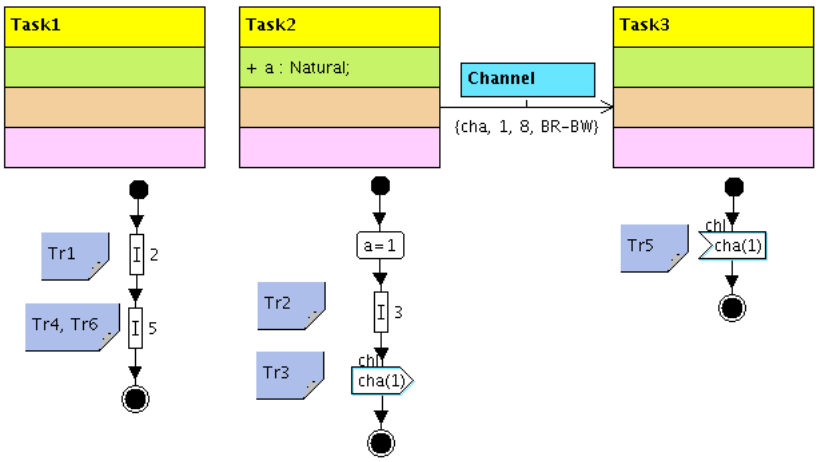


Figure 5.4: Example Scenario: Application

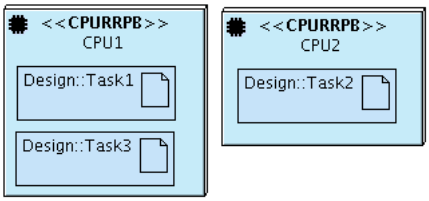


Figure 5.5: Example Scenario: Architecture

5.6 Implementation Issues

5.6.1 Simulator Architecture

The architecture of the simulator was conceived with a twofold separation of concerns in mind: application from architecture on the one hand and communication from computation on the other hand. In the view of future research work targeting for instance energy consumption aspects, modification and enhancement of the environment should be feasible with little effort. For this reason, the architecture is structured in five layers, representing tasks, communication among tasks, execution hardware, communication hardware and the simulation kernel. Section 5.5.2 elaborates further on the different layers. The simulation environment was developed in C++ in an object oriented fashion, so that the architecture is structured in terms of classes and interfaces. The adequate use of inheritance, polymorphism and templates paves the way for a hassle-free integration of new application or architecture components. This section sheds light on the classes at each of the simulation layers and their interrelationships.
Figure 5.6 provides a quick overview of the layered simulator architecture. The lowermost row in Figure 5.6 depicts global interfaces which may be used at any layer. Inheritance relations among classes are denoted with arrows which are drawn in the same color in case they stem from a common base class . Figure 5.6 is far from being exhaustive as lots of classes have been left out for the sake of simplicity.

5.6.1.1 Interfaces

Interfaces generalize functionality being available in different contexts across all simulation layers. More particularly, a standardized procedure governs the way in which workload (e.g. transactions) are passed between components, benchmarking data is exchanged, simulation states are saved and recovered. This implementation choice apparently serves the goal of homogenizing the architecture and so facilitating debugging and maintenance.
As its name suggests, *WorkloadSource* constitutes an interface of classes providing a workload to other objects. Tasks for instance provide a workload to their scheduler. Scheduler may provide a workload to other schedulers in a scheduling hierarchy or directly to processing elements such as CPUs. The *Workload* interface comprises methods such as *getNextTransaction* to communicate the scheduling decision to the callee, *getPriority* to determine the priority of the workload source and *schedule* which initiates a new scheduling round. The *TraceableDevice* interface allows documentation functions to query an object for benchmark data. For instance, at the end of a simulation run the interface is used to extract performance key figures from CPUs and buses.
The *Serializable* interface is implemented by all classes that make up the overall simulation state (such as *TMLCommand*, *TMLChannel*). The *writeObject* and *readObject* methods transform relevant attributes into bit streams and vice versa. This functionality is crucial for saving and recovering encountered simulation states which is needed to consecu-

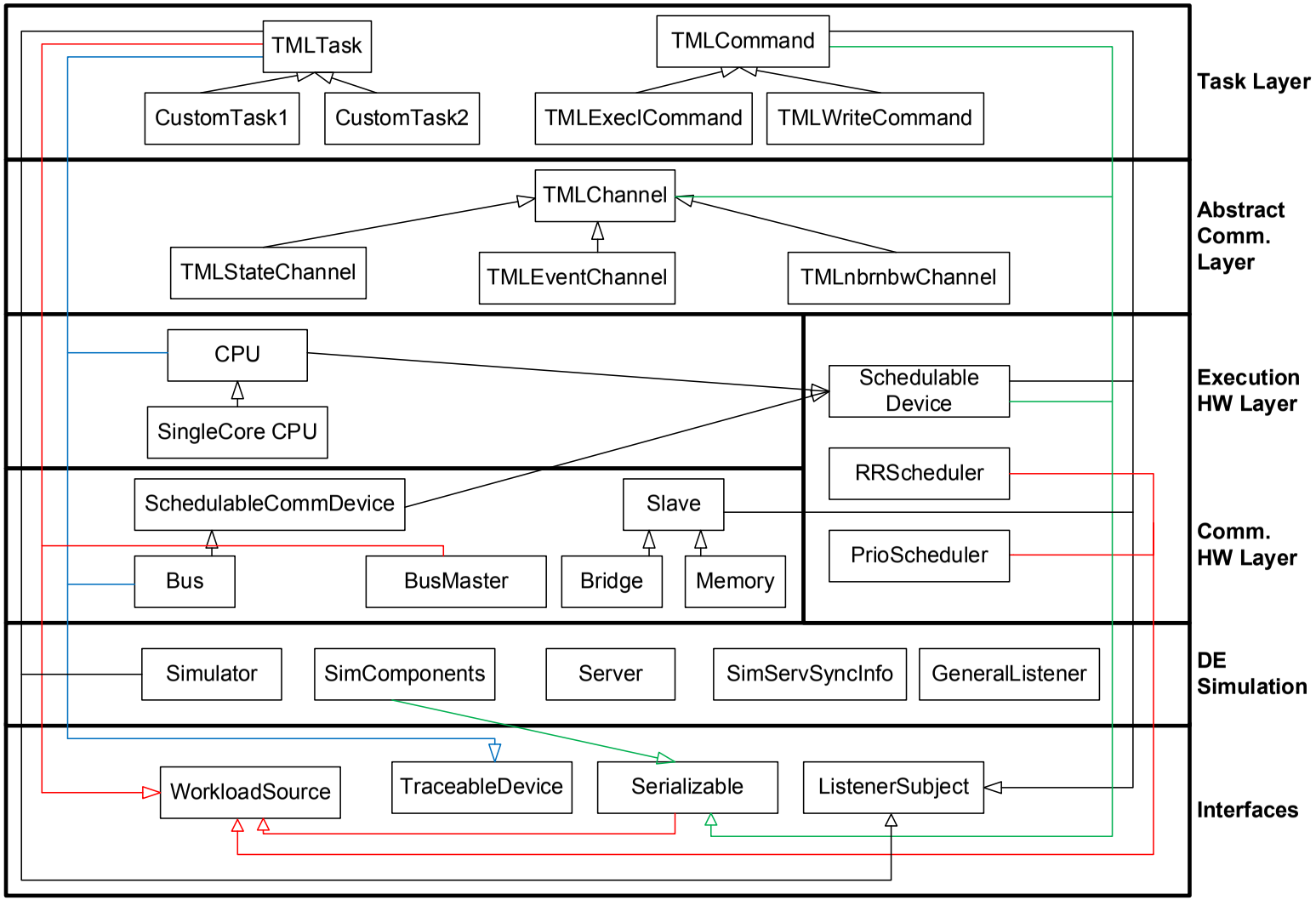


Figure 5.6: Simulator architecture

tively explore several branches of control flow. The generated bit streams may either be saved in the RAM or on the hard disk.

5.6.1.2 Task Layer

Attention is now drawn to the classes within the layered structure. At the top of it, *TMLTask* and *TMLCommand* classes constitute the *Task Layer*. Its emphasis is placed on DIPLODOCUS task and operator semantics and it provides a common logical framework for all imaginable DIPLODOCUS tasks. The broad variety of DIPLODOCUS operators is transparent to the rest of the simulation environment.
All specific DIPLODOCUS application operators inherit from the *TMLCommand* class. *TMLTask* serves as an abstract class and therefore cannot be instantiated directly. Concrete task classes are generated from the graphical UML model by a C++ code generator and they have *TMLTask* as a common subclass. These customized task classes accommodate references to all commands that are contained in the DIPLODOCUS task. Their constructor establishes the control flow by instantiating and chaining *TMLCommand* objects accordingly. To satisfy the encapsulation principle, tasks variables are defined locally in a task class derived from *TMLTask*. However, commands operate on local task variables, to check for conditions, evaluate the variable length of operators, carry out actions, etc. For that purpose, *TMLCommand*s are given pointers to functions internal to a task, which in turn have access to the local variables.
TMLCommand defines an interface comprising the following methods: *prepare* initiates the *Evaluation Phase*, in which commands are executed until the first transaction is triggered (as explained in Section 5.5.2). *execute* enters the *Update Phase* which aims at updating the state of simulation components according to the transaction selected by the simulation kernel. *getCurrTransaction* is inherited from the *WorkloadSource* interface and invoked by CPU schedulers to obtain the next transaction of a task. *prepareNextTransaction* generates a new transaction with respect to the progress of the command and its semantics. A subclass of *TMLCommand* usually has to implement *prepareNextTransaction* and *execute* as these two methods highly depend on the purpose of the command. *TMLCommand* further manages a break point which can be used to halt the simulation under specific circumstances.

5.6.1.3 Abstract Communication Layer

The *Abstract Communication Layer* implements the separation of communication and computation at application level. The different DIPLODOCUS communication media are classified in an inheritance hierarchy according to their semantics such as blocking characteristics and order relation (e.g. FIFO). Classes residing at this layer have a common super class called *TMLChannel*. This class subsumes all available means of communication/synchronization in DIPLODOCUS, be it Channels, Events or Requests. The *TMLChannel* interface is characterized by four methods, two belonging to the *Evaluation Phase*, and another two belonging to the *Update Phase*. *testRead* and *testWrite* check

whether a transaction is runnable according to synchronization constraints (channels full/empty, events available in FIFO or not,etc.). If needed, the virtual length of a transaction may be reduced. Consequently, this has to be accomplished during the *Scheduling Phase* before CPU schedulers query tasks for transaction proposals. The methods *read* and *write* finalize a communication by updating internal channel variables such as the content and internal FIFOs. This is of course accomplished in the scope of the *Update Phase*. *TMLStateChannel*, *TMLEventChannel* and *TMLnbrnbwChannel* count among the direct subclasses of *TMLChannel*. *TMLStateChannel* is the super class of all DIPLODOCUS channels presenting a blocking behavior. As the name suggests, *TMLEventChannel* models the FIFO based synchronization scheme of DIPLODOCUS events. *TMLnbrnbwChannel* is a dummy class to account for the simplest of all DIPLODOCUS channels which is never blocking.

5.6.1.4 Execution HW Layer

The *Execution HW Layer* constitutes an extensible skeleton for components for execution of DIPLODOCUS tasks, such as CPUs, Hardware Accelerators, and perhaps also DMAs. It satisfies the separation of communication and computation at architecture level. For the time being, it only accommodates the *CPU* class and the derived class *SingleCoreCPU*. As stated in section 5.4.2, HWA accelerators are modeled with a properly parametrized CPU component. The *CPU* base class could also serve as common interface for other more specific components like DMAs or a refined model of a hardware accelerator. *SingleCoreCPU* acts as a model of an operating system together with a model of a single core CPU. It mainly implements the inherited interfaces *Traceable*, *Serializable* and *Schedulable*. The *truncateAndAddNextTransAt* methods realizes the transaction truncation functionality of the kernel explained in Section 5.5.4. Even if multi core CPUs do not directly have a counterpart in the class hierarchy, the simulation kernel offers the possibility to treat a set of *SingleCoreCPU* objects as such.

5.6.1.5 Schedulers

Classes related to the scheduling functionality are common to both Execution and *Communication HW layer*. This architectural choice allows to reuse algorithms once defined on both layers and considerably alleviates the integration of new policies. The *schedulableDevice* interface subsumes functionality to manage schedules and to derive a graphical or textual representation for the latter. The *schedule* method for instance is unique to each scheduling algorithm and describes the scheduling policy. The *addTransaction* method concludes a simulation round by definitively adding a transaction to the schedule of a component (*Update Phase*). *getNextTransaction* is invoked by the component owning the scheduler to ask for the result of the scheduling procedure. The *SchedulableDevice* interface is implemented by the *CPU* and the *Bus* class. *RRScheduler* and *PrioScheduler* mainly override the *scheduling* method to model a round robin, time slice based policy and a fixed priority policy respectively. Due to the transaction based simulation, the

local simulation time of active components (CPUs and Buses) differs. For that reason, a scheduler has to discriminate tasks that became runnable in the past from task that will become runnable in the future with respect to its local time. Runnable times in the past are handled in compliance with the scheduling policy. If all runnable times lie in the future, the task which becomes runnable first is selected by the scheduler. This simply stems from the fact that the scheduler cannot anticipate transactions in the future.

5.6.1.6 Communication HW layer

Below the *Execution HW Layer*, the *Communication HW layer* resolves resource contention on shared communication media. The *SchedulableCommDevice* extends the *Schedulable* interface with mainly two methods: *getBurstSize* and *registerTransaction*. The former defines the size of the largest, atomic bus transaction referred to as burst. The latter notifies the arrival of a new resource request and is supposed to invalidate the scheduling decision of the component. *Buses* perform scheduling in a lazy fashion, that means when their *getNextTransaction* method is invoked and under the condition that the previous decision has been invalidated. A *Bus* defines the methods required by the interfaces *SchedulableCommDevice* and *Traceable*. Moreover, it is able to calculate the physical duration of bus transactions. The *BusMaster* class plays the role of the mediator between one CPUs and potentially several *Bus* objects. The number of instantiated bus objects per DIPLODOCUS Bus amounts to the number of channels the user has specified for the DIPLODOCUS bus. A *BusMaster* has to assure that a single transaction is never granted more than one bus channel at a time. This is mainly achieved with the method *accessGranted*, which queries all buses for their next transaction (via the *getNextTransaction* method) in the order of ascending *endSchedule* times of the buses. *endSchedule* thereby denotes the end time of last transaction executed by the respective bus. The *getNexTransaction* method of the bus master, in turn called by bus schedulers to determine the workload, only returns a transaction if it has not yet been scheduled on another bus channel.
The *Slave* interface contains methods permitting slaves to delay transactions (method *CalcTransactionLength*) and to add a transaction to the local schedule of the device (method: *addTransaction*). For the time being, solely *Bridges* and *Memories* implement the slave interface.

5.6.1.7 DE Simulation

The *DE Layer* finally embraces all general functions relevant for simulation (class *Simulator*) and is to a large extent independent of the semantics of architecture and application. However, for performance reasons, the simulator was built on the assumption that point to multipoint communication does not exist among tasks. The assumption could easily be relaxed with slight performance penalties.
Furthermore, the layer hosts the management of simulation components (class *SimComponents*), output of traces in various formats (class *Simulator*, presented in Section

7.4), coverage enhanced exploration of DIPLODOCUS applications (class *Simulator*, see chapter 6) and thread safe synchronization of the integrated server module with the IDE module (class *SimServSyncInfo* and *Server*, see Section 7.5).

5.6.2 An exemplary simulation run

To illustrate the simulation concept, this section presents the interaction of simulation components with a simple example. We follow the execution of Task2 in Figure 5.4, that only consists of a sequence of an Action command, an Execi command and a write command. The architecture comprises a CPU, onto which the task is mapped, a bus and a memory which are supposed to handle the data transfer issued by the write command.

The sequence diagram in Figure 5.7a depicts the *Evaluation Phase* of the Action command and the *Evaluation* and *Scheduling Phase* of the Execi command. The simulator first queries the task for the current command (Step 1) and then sends a *prepare* message to the latter (Step 2). As the Action command does not generate a transaction, the corresponding action function can be invoked (Step 3,4) right away within the *prepareNextTrans* method. The action function incorporates the action to be carried out (variable assignment, etc.) and belongs to the scope of the task. Then control flow is passed on to the prepare sequence of the Execi command (Step 5,6). As the length of the command could be a sophisticated function of task variables, it has to be evaluated within a dedicated function of the task (Step 7). The process culminates in the creation of a new transaction, whose attributes *virtualLength* and *startTime* are initialized with the length of the DIPLODOCUS command and the *finishTime* of the previous transaction of the task respectively.
Thereafter, the *Scheduling Phase* is initiated by *schedule* message (Step 9) that the kernel sends to the CPU. The CPU in turn forwards the message to its scheduler (Step 10) and prompts it for the scheduling result (Step 11). The scheduler sends a query for the current transaction to each registered task (Step 12) and the task finally forwards it (Step 13) to the current command. After the CPU has calculated the *startTime*, the *duration* and the penalties of the transaction based on its internal schedule and parameters (Step 14), the transaction is modified accordingly (Step 15).

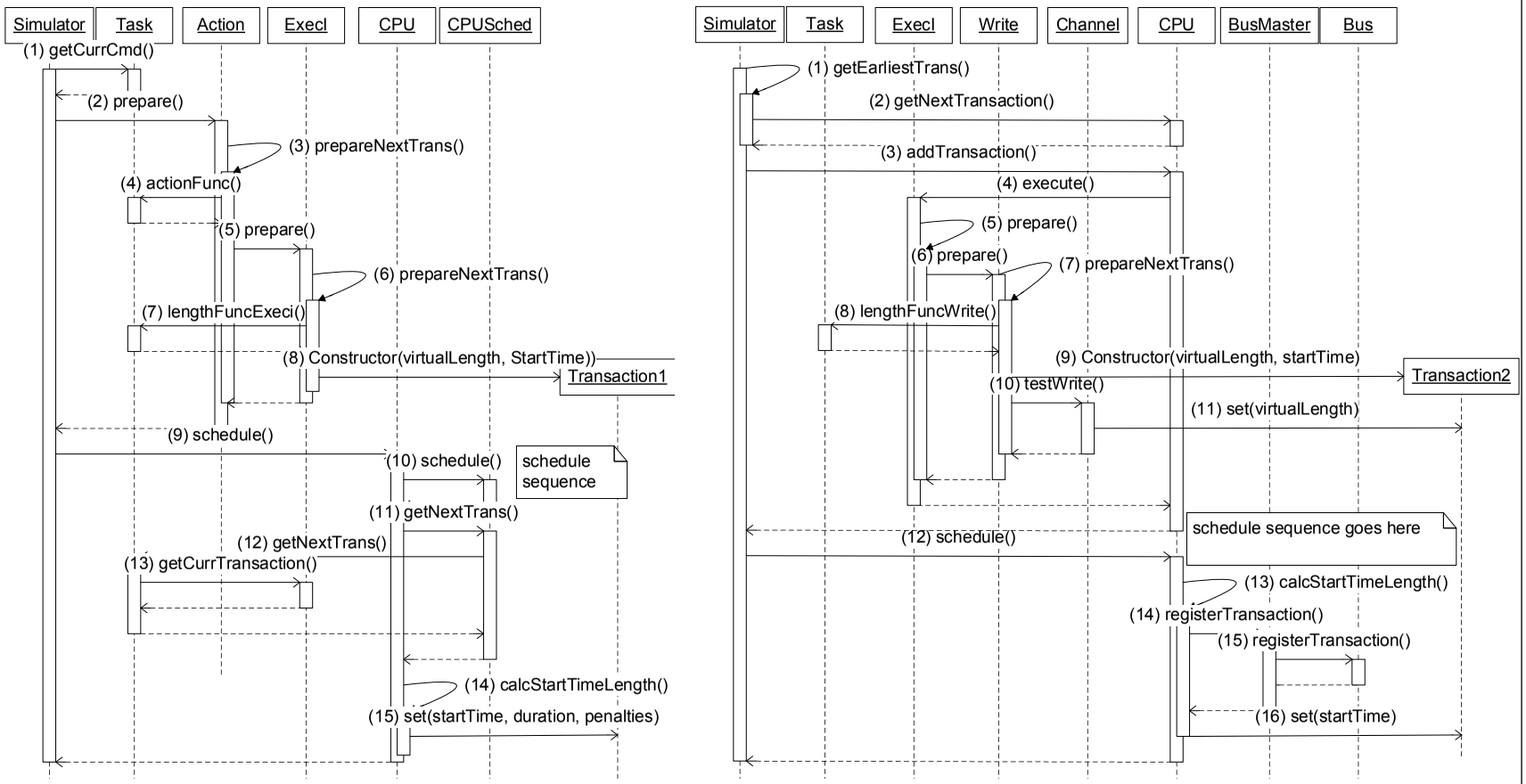
Figure 5.7b illustrates the *DE/Update Phase* of the Execi command followed by the evaluate and *Scheduling Phases* of the Write Command. To find the transaction that terminates earliest, the Simulator queries each CPU for its next transaction (Step 1,2) within the *getEarliestTrans* procedure. The CPU proposing this transaction then receives the order to add it to its local schedule (Step 3). The CPU in turn invokes the *execute* method (Step 4) of the Execi command which makes the command update its internal state. This concludes the *Update Phase* and a *prepare* message forwarded to the write command (Step 5,6) announces the recurrence of an *Evaluation Phase*. The *prepareNextTransaction* (Step 7) is as usual in charge of spawning a new transaction. As the number of samples could

be a complex expression, it is evaluated in the scope of the task (Step 8), in analogy to the length of the Execi command. The write command also initializes the *virtualLength* and *startTime* of a newly created transaction (Step 9). So far, the procedure is identical to the *Evaluation Phase* of the Execi command. However Step 10 and 11 are specific to data exchange (read, write) and synchronization (Wait Event, Send Event) commands. The channel is informed about a write command awaiting execution (*testWrite* message) and determines the *virtualLength* of the transaction as a function of its size and filling level. The size could also be set to zero in case the channel is full, which makes the transaction invisible to schedulers. Thereafter, the same scheduling sequence is encountered as for the Execi command (Figure 5.7a, Steps 9-13). In Figure 5.7b this part has been left out, except for the *schedule* message (Step 12). In the scope of the *calcStartTimeLength* method (Step 13), the CPU advertises the transaction to the BusMaster (Step 14) which in turn invalidates the current schedule of the bus (Step 15). Finally, the CPU may postpone the *startTime* of the transaction, depending on its schedule (Step 16).

Figure 5.8 shows the completion of our scenario: the write command goes through the *DE* and *Update Phase*. Steps 1 and 2 are similar to what was encountered for the Execi command (compare Figure 5.7b). The simulator again searches for the runnable transaction with the earliest *finishTime*. This time, upon reception of a *getNextTransaction* message (Step 2), the CPU has to make sure that the write transaction is really granted access to the bus. Therefore, it issues a *grantedAccess* message (Step 3) to the bus master. The latter forwards the query to the bus (Step 4), which in turn realizes that its schedule is obsolete and initiates (Step 5) a scheduling operation at the bus scheduler (Step 6). When it comes to the *Update Phase*, *addTransaction* messages travel from the simulator via the CPU and the bus master to the bus (Step 7-9). This laborious resolution is necessary to allow CPUs to comprise several bus masters, and bus masters to be connected to several buses (recall: one bus object is instantiated per communication channel of a DIPLODOCUS bus). Thereafter, the memory is involved in the calculation of the *duration* of the transaction, to account for memory access delays (Step 11). Last but not least, the inner state of the write command and the channel have to be updated. To this end, the simulator sends an *execute* message (Step 12) to the write command and the latter issues a *write* message to the channel (Step 13). Again, this concludes the *Update Phase* and the *Evaluation Phase* for the command following the write command is entered (Step 14,15).

5.6.3 Simulation event dispatching

The architecture depited in Figure 5.6 relies on the observer pattern to separate simulation logic from peripheral logic that is responsible for the evaluation of simulation events. It should be noted that simulation events indicate actions taken by the simulation engine and should not be confused with DIPLODOCUS events. In Figure 5.6, all classes implementing the *ListenerSubject* interface are considered as simulation event sources. That means that they provide facilities to register so called event listeners which must



(a) Action Command and Execi Command

(b) Execi Command and Write Command

Figure 5.7: Example Scenario: Sequence Diagrams

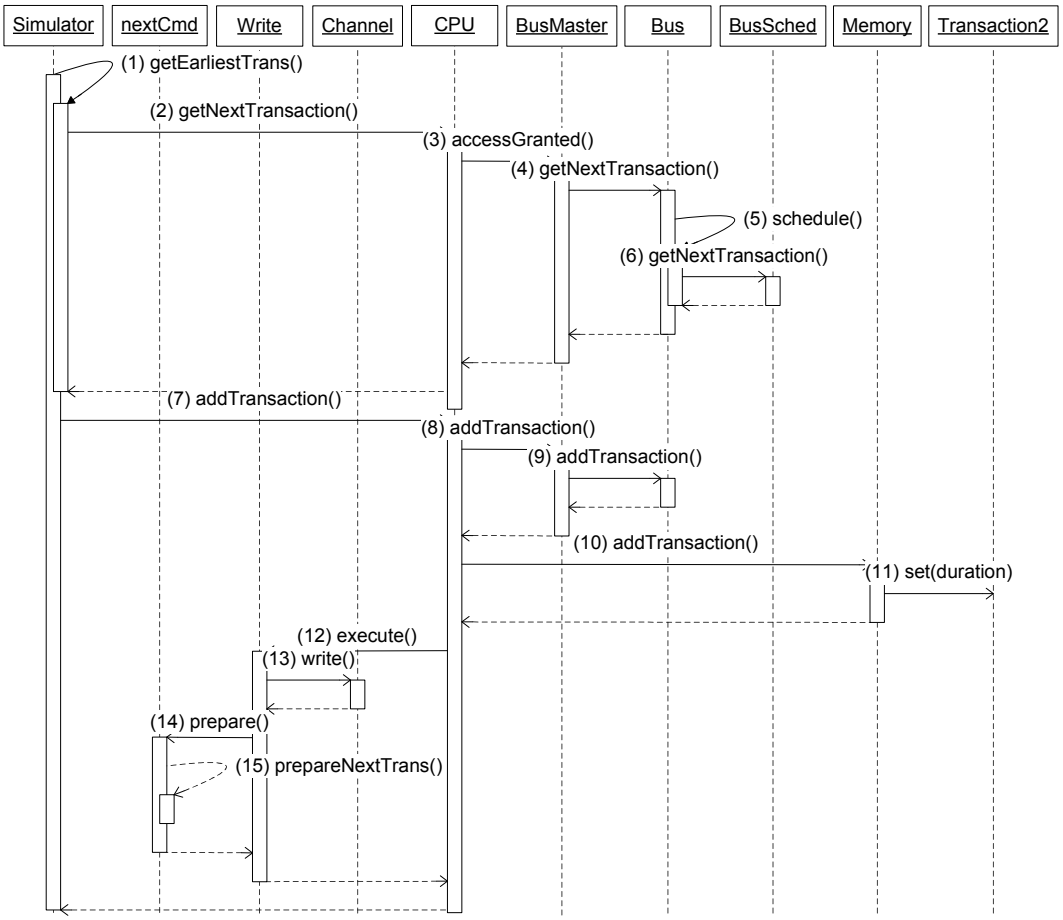


Figure 5.8: Completion of the Write Command: Sequence Diagram

Operation	1 CPU	2 CPU
Execi	1.55	1.52
Send/Wait	1.46	1.41
Read/Write	0.86	0.79

Table 5.1: Simulator speed in MTrans./sec

inherit from the *GeneralListener* interface. Event listeners react to predefined events by overriding the respective method of the base class. *ListenerSubject* may generate a subset of the following events:

- *simulationStarted*: is notified once per simulation run upon the start of the simulation
- *simulationStopped*: is notified once per simulation run upon the end of the simulation
- *timeAdvances*: is notified whenever a transaction is scheduled by the simulation kernel
- *taskStarted*: is notified once when the first transaction of a task is executed, if the task receives requests the event is notified several times
- *taskFinished*: is notified once at the end of a task if the task does not receive requests
- *transExecuted*: is notified upon every executed transaction on a particular device (Bus, CPU,etc.)/application entity (task, command,etc.)
- *commandEntered*: is notified as soon as control flow reaches a command
- *commandFinished*: is notified when the last transaction of a command is executed, before control flow passes on
- *commandExecuted*: is notified upon execution of every transaction of a command, except for the last one
- *commandStarted*: is notified upon execution of the first transaction of a command

5.6.4 Experimental results

We consider three of the most prevalent types of operations to evaluate the performance of the simulation engine: event Send/Wait operations, Read/Write operations on channels and Execi operations. From our experience, these operations make up the lion's share of DIPLODOCUS applications together with Choice and Action commands. The latter are implicitly considered as our operators are nested in a main loop which is

internally represented with structures similar to Choice and Action commands. Table 5.1 specifies the simulation speed for an application comprising two tasks which only perform the given operations in a loop. Speed is measured in millions of transaction per second of host processor time. The task set has been mapped onto an architecture composed of 1 CPU and 2 CPUs respectively, which are represented in two separate columns in the table. For experiments with read/write operations, the architecture was extended with one memory and one bus. As expected, Execi operation are the less costly because they do not involve any synchronization. Send and Wait transactions are in between the latter and Read/Write transactions, which are most time consuming as communication media need to be arbitrated. The simulation on an architecture with two CPUs is more costly than on one with a single CPU due to additional scheduling overhead.
The simulation speed in terms of $\frac{cycles}{sec}$ may be calculated as the product of the above mentioned simulator dependent part in $\frac{trans}{sec}$ and the average transaction length in $\frac{cycles}{trans}$. A strength of this simulation policy is consequently that the simulation speed increases with the abstraction level (or granularity in other terms) of the model. This was not the case for the ancient SystemC simulator, existing prior to this thesis. Figure 5.9 provides evidence that simulation time increased more or less linearly with the average transaction length. This experiment is conducted with a task set comprising two tasks: Task1 first sends an event of type evt1 to Task2, then performs an Execi of length l and finally waits for an event of type evt2. Task2 carries out the complementary operations: it waits for the reception of event evt1, then also performs en Execi of length l and finally sends and event of type evt2. The commands in both tasks are iterated a million times. Figure 5.9 reveals that the length l of Execi commands is varied between 1 and 10. Even in the worst case of an average transaction length of $l = 1$, the new simulator outperforms the one based on SystemC by factor 10.
For all measurements, output capabilities of both simulators have been disabled in order to avoid a distortion of results due to file I/O operations. The time consumed by the initialization procedure of both simulation environments is neglected so as to measure the real simulation time. The measured execution time may be subject to noise caused by the other tasks running on the operation system. Therefore, the given results constitute an average of several time measurements. The noise should thus equally distort the results for both simulators. The performance study was conducted under Linux, Fedora Release 11, on a Intel Dual Core CPU 2.7 GHz with 4 GB RAM.

5.7 Conclusions

In this Chapter, a simulation approach especially suited for high level models of Systems-on-Chip was presented. The reason why we did not rely on the wide-spread SystemC standard is twofold.

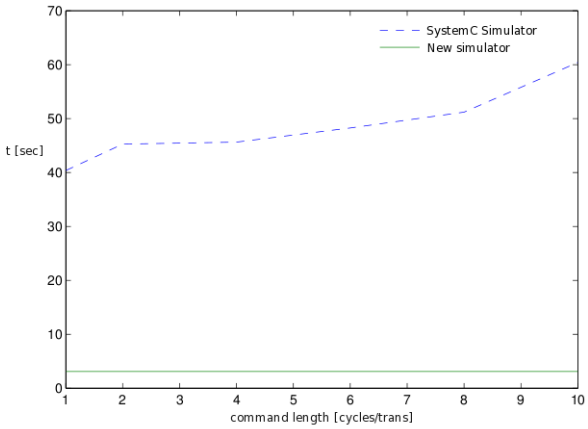


Figure 5.9: Simulation time as a function of the average transaction length

1. On the one hand, it was argued that SystemC concurrency primitives should be used scarcely due to their impact on simulation performance. This insight calls for a customized management of DIPLODOCUS tasks, that way reducing the number of SystemC threads compared to the number of active components such as CPUs.

2. On the other hand, it was shown that DIPLODOCUS execution semantics paves the way for further improvements of the conventional DE simulation approach. Simulation of high level models can renounce many features provided by the standard SystemC kernel, such as sensitivity to signals, repeated execution of threads until a steady state is reached, management of different concurrency primitives, an explicit representation of events, event queues as well as the creation and cancellation of events.

In conclusion, the actual parts of the SystemC implementation reusable for DIPLODOCUS models would be restricted to a small subset of the kernel. Given that the existing implementation therefore would have to be analyzed thoroughly, we believe that our inhouse development made us not only realize a benefit in terms of simulation performance, but also in terms of development time. The performance figures reveal not only a notable performance gain, but also suggest that abstraction in the model is directly leveraged to speed-up simulation. The simulation approach is merely specific to high level models in general, not to either DIPLODOCUS or UML models in particular. It could be adapted to other input models by providing a dedicated code generator. The code generator for DIPLODOCUS is covered in chapter 7.

Chapter 6

Extending Simulation coverage

6.1 Introduction

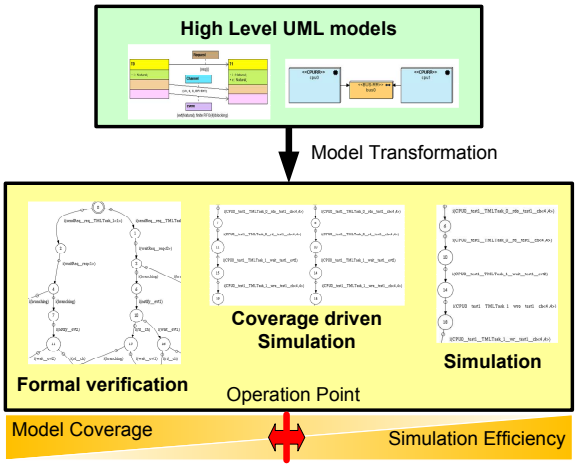


Figure 6.1: Varying Model Coverage in DIPLODOCUS

Since formal verification on low-level models faces the state explosion problem, it is merely applied to subparts of the system where data has been abstracted to its mere presence or absence. This limitation is pushed further when raising the abstraction level. Still, it remains a major obstacle for the verification of medium sized system level models. Furthermore, representing high-level mapping models (e.g. models comprising an application and architecture) is extremely cumbersome in formal languages. Methods such as DIPLODOCUS have been proposed to transform graphical high level models (UML, etc.) into a representation amenable to model checking. To this end, UML models must be endowed with a formal semantics. Nonetheless, we experienced that the transformation of sophisticated mapping models results in complex syntactical

structures, often pushing both UPPAAL [24] and CADP [38] model checkers to their limits.
Moreover, even if the model checker is able to handle the specification, system space coverage cannot be varied: formal verification is exhaustive by definition and consequently model-checkers are conceived to explore the entire state space. For that reason, varying the coverage of the source model requires to reconfigure the model transformation algorithm and to regenerate the formal model. However, from a designer's perspective, it would be desirable to generate an executable model once and to subsequently traverse an interesting fraction of its state space. The design space may be pruned with the aid of conventional coverage criteria (with respect to covered branches, statements, tasks, conditions,etc.), expert knowledge provided by the user (e.g. potentially critical parts of the control flow to the environment), or heuristics taking into account (non-) functional properties. In any case, the objective is to allow for taking decisions dynamically, at simulation runtime. That way, the developer may trade off more easily simulation efficiency against coverage without regenerating the formal model. The operation point thus ranges between the two extreme cases of formal verification and simulation, which is denoted with the slider in Figure 6.1). Clearly, when moving away from exhaustive formal verification, proves in a mathematical sense cannot be performed any more.
To address the aforementioned trade-off, we propose a novel way to enhance simulation coverage of high level models based on model checking and static program analysis techniques. A simulator steamlined for high level models of SoC was introduced in Chapter 5. It already enables the developer to manually explore several branches by saving the simulation state at a decision point and getting back to it later. However, at the stage described in Chapter 5, it lacks adequate state storage and comparison techniques which are essential for merging logically equivalent execution paths. This chapter points out remedies to this shortcoming. DIPLODOCUS mapping models are statically analyzed to spot elements not being part of the state vector so as to speed up the identification of recurring system states. Section 6.3 elaborates on that procedure. Furthermore, section 6.4 demonstrates a way to localize points in the model where executions are likely to be joined and therefore states must be compared. In section 6.5, insights are given into the implementation of for instance dependencies relations and state hashing. Methods for the identification of recurring system states are addressed in section 6.5.5.

6.2 State Space of DIPLODOCUS models

The state space of DIPLODOCUS applications comprises all possible interleavings of task executions. It is solely constrained by inter-task synchronization (events and requests) and data dependencies (blocking channels). As depicted in Figure 6.2, the state space is spanned by the current position within a task, local task variables and the state of inter task communication primitives. According to Chapter 5, a command may be broken down into several consecutively scheduled transactions. The current position in a task is thus the combination of the current command and its progress. The progress of

a command amounts to the sum of the virtual length of all completed transactions. The state of channels where either read or write operations are blocking can be characterized by the number of samples currently stored in them. If both read and write operations are non-blocking, channels can be considered as stateless. Each sent events or requests may convey parameters. Therefore, parameters of all pending (sent but not yet received) events/requests must be recorded in a separate data structure. For events based on an infinite FIFO and requests (by default based on an infinite FIFO), state vectors are unbounded.

According to the definition in section 3.2, a state makes the system's history irrelevant and permits to deduce unambiguously the response to future stimuli. In DIPLODOCUS models, the above mentioned elements of the state vector serve this goal. Past state transitions of the architecture model are implicitly contained in this state vector, as the architecture has restricted the set of possible behaviors of the application model. While application related information is sufficient for the detection of recurring system states, it is unsuitable for restoring performance measures. For example, the simulator constantly records the number of transactions processed on HW components, the local time of HW components, the execution time of tasks, the overall contention delay experienced for shared resource allocation, etc. These values do not impact future system behavior and therefore are not part of the system state vector. However, performance values must be restored when getting back to previous system states. This fact is acknowledged by an extended vector, containing both intermediate performance figures and the system state.

Fortunately, not all partial orders of concurrent actions of the application model have to be explored as soon it is bound to a mapping. Indeed, the mapping of application tasks onto an architecture constrains the state space due to shared resources like processors, buses, etc (cf. Constraints arrow in Figure 6.2). However, as data dependent behavior is abstracted with non-deterministic operators, yet several execution traces exist. (The behavior of the architecture is assumed to be deterministic, compare section 5.4). Traces may significantly differ in terms of non-functional properties like execution time and resource usage. and in whether they satisfy functional requirements. It may therefore be important to explore more than one possible branch. To prevent equivalent execution paths from being explored more than once, representations of encountered states have to be maintained, similar to those in Model Checkers like SPIN [53]. The main objective of this chapter is to suggest techniques essential to the realization of the coverage selector shown in Figure 6.2 for UML models of SoC. Deriving appropriate coverage criteria for DIPLODOCUS models as mentioned in section 6.1 is however out of scope of this work. The challenge tackled in this thesis is twofold: it is crucial to minimize both the size of particular state vectors and the number of times the latter have to be stored and compared.

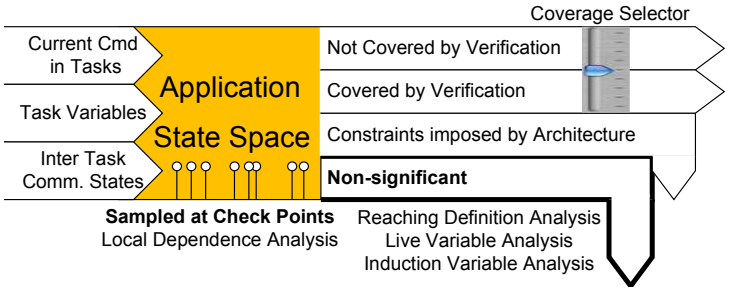


Figure 6.2: State Space Exploration Concept

6.3 Static Analysis of DIPLODOCUS applications

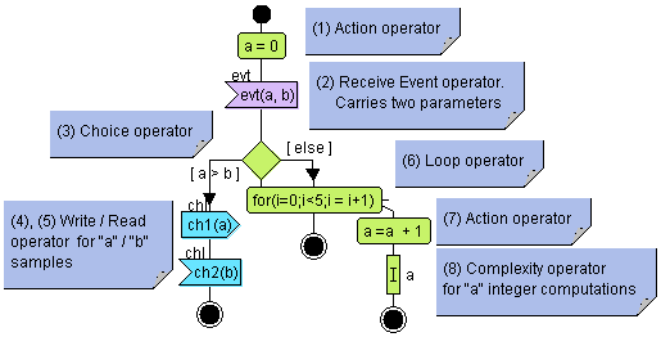


Figure 6.3: Running example of an Application Model

This section covers static analysis methods applied to DIPLODOCUS models in order to to identify significant state variables at each point in an activity diagram. Explicit state representations are greedy in terms of memory. A thorough analysis of applications is necessary to minimize information required to uniquely describe system states. Furthermore, system states encountered during simulation have to be compared to all previous ones in order to merge similar execution paths. Reducing the footprint of state vectors also comes with the positive side effect of cutting down time needed for comparison.
The bold arrow on Figure 6.2 illustrates the outcome of the two of the three pivotal techniques presented in this section: *Reaching Definition Analysis* and *Live Variable Analysis*. Informally, *Reaching Definition Analysis* brings out reaching definitions of a DIPLODOCUS operator. A reaching definition is another operator comprising a variable assignment that reaches the former operator. Thereby, the notion of reaching refers to a possible path in control flow, be it conditional or not. *Live Variable Analysis* determines variables considered to be live in a DIPLODOCUS operator, e.g. if they may potentially be read before being redefined.
The latter two analysis methods are commonly accomplished by compilers. These techniques belong to the realm of machine independent optimizations [7], as they only

require knowledge of an algorithm, regardless of the underlying execution platform. Optimizations of this kind have been around for a while and will therefore be described very concisely. However, *Local Input Dependence Analysis* is somewhat particular to the DIPLODOCUS model of computation and is therefore covered in more detail. It detects so called *Check Points* in a task where traces are likely to branch or join entailing that states must be stored and compared (cf. sampling symbols in Figure 6.2).

The diagram depicted in Figure 6.3 serves as a running example throughout this section. The contained DIPLODOCUS operators are henceforth referred to by their numbers indicated inside of the memo symbols in the diagram. The application consists of an initialization part which first sets the variable a to zero and then receives an event carrying two parameters. The parameters are saved in the task variables a and b. Depending on the condition $a > b$, control flow is either directed to a sequence of *Write* (a samples) and a *Read* command (b samples) or to a loop, whose body contains a variable assignment and a subsequent symbolic operation (*Execi a*).

6.3.1 Basic blocks

A decomposition of the control flow into *basic blocks* is commonly used in static program analysis. The purpose of basic blocks is to abstract a DIPLODOCUS task with respect to the domain of a particular analysis. The domain is characterized by a set of possible states observable at a point in the task. As a concrete example, Live Variable Analysis operates on the power set of all variables used in the task, and Reaching Definition Analysis on the power set of all definitions. Every basic block is assigned two data flow values out of these sets, revealing for instance which variables are considered to be live and which definitions reach the block [7].
As it will be shown in dedicated sections (6.3.2, 6.3.2), data flow values being valid before and after a basic block are related to each other by a transfer function, abstracting the operations of that basic block. Forward analysis differs from backward analysis in that the transfer function is defined in the inverse way. Moreover, control flow (e.g. the interconnections of basic blocks) imposes constraints on data flow values of adjacent blocks.

Basic blocks are generally defined as maximal sequences of consecutive instructions that have exactly one entry and no internal branching. In the context of DIPLODOCUS, an analysis is always carried out with respect to a DIPLODOCUS task t, and so all of the following definitions are also specific to a task t. To simplify matters, the index t is however omitted. A basic block s is mainly defined by the following characteristic functions:

- $Def\,(s) = \{v | v \text{ is a variable defined/modified in } s\}$, $\left|Def\,(s)\right| \leq 1$, is the set of defined variables. To simplify some expressions, the number of defined variables is lim-

ited to 1. As a consequence, DIPLODOCUS operators modifying more than one variable (e.g. Wait Event) translate into one basic block per variable.

- $ID(s)$ is the basic block ID function, if a DIPLODOCUS operator is mapped onto several basic blocks $s_1 \dots s_x \dots s_n$, then $ID(s_x) = x$.
- $Ref(s) = \{v | v$ is a variable referred to but not modified in $s\}$, is the set of read variables
- $Comm(s) = \{c | c$ is a communication primitive (Channel, Event, Request) used in $s\}$, $0 \leq |Comm(s)| \leq 1$, is the set of communication primitives
- $Succ(s) = \{s' | s'$ is a successor of $s\}$, is the set of successors of the basic block. Unless otherwise defined, DIPLODOCUS operators correspond to one basic block. The successor relationship is established in exactly the same way as in the DIPLODOCUS model.
- $Pred(s_2) = \{s_1 | s_2 \in Succ(s_1)\}$ is the set of predecessors of s_2
- $Path(s_1, s_2) = \begin{cases} true & \text{if} \quad s_1 = s_2 \\ \bigvee\limits_{s_t \in Succ(s_1)} Path(s_t, s_2) \vee Path(s_2, s_t) & \text{otherwise} \end{cases}$
  is the path function that evaluates to true if there exists a path between two basic blocks s_1 and s_2.

As stated before, we consider each operator of a DIPLODOCUS activity diagram as a basic block, except for *Loop* and *Wait Event* operators. *Loop* operators are decomposed into three basic blocks, namely into an initialization, an increment and a condition block. The initialization block is connected to the condition block, the condition block to the first block of the loop body and to the first block succeeding the loop, the loop body to the increment block and finally the increment block back to the condition.
In case DIPLODOCUS operators assign values to several variables, each of these assignments is handled in a dedicated basic block. The resulting basic blocks are connected consecutively and carry different IDs. The connections between all other basic blocks are established in the same way as in the corresponding activity diagram.

In the following, the initial definition of a basic block is complemented with some notions essential for formalizing static analysis stages in later sections. Therein, V denotes the set of all variables of a task and S is the set of all basic blocks of a task:

- $In(s) = \{(s', v) | v$ was last defined at s' before the execution of s$\}$, is the set of incoming definitions, i.e. $v \in Def(s')$, and there is a path from s' to s, on which there is no other node s'', where $v \in Def(s'')$ except s'
- $Out(s) = \{(s', v) | (s', v) \in In(s) \wedge v \notin Def(s)\} \cup \{(s, v') | v' \in Def(s)\}$, is the set of outgoing definitions

- $Gen(s) = \begin{cases} \{(s,v) | v \in Def(s)\} & \text{if} \quad Def(s) \neq \emptyset \\ \emptyset & \text{if} \quad Def(s) = \emptyset \end{cases}$ is the generating set which contains (s,v) if the basic block s defines a variable v
- $Kill(s) = Out(s) \setminus In(s) \setminus Gen(s)$ is the set of definitions killed by s
- $Dep(s)$: Boolean value indicating whether the basic block depends on non deterministic choices
- varFunc: $\{(s,v)\} \rightarrow \{v\}$ is a projection of a set of basic block/variable pairs (s,v) on the set of variables v which are contained in the preimage vectors
- blockFunc: $\{s,v\} \rightarrow \{s\}$ is a projection of a set of basic block/variable pairs (s,v) on the set of basic blocks s which are contained in the preimage vectors

6.3.2 Live Variable Analysis

The intention of this analysis is to find out whether the value of a variable x at point p could be used along some path in the control flow starting at p. If so, the variable is significant and thus constitutes the system state at point p. Otherwise it can be safely neglected at point p, and the corresponding assignment can be omitted. The transfer equations for Live Variable Analysis are defined as follows:

- $VarIn, VarOut \in \wp(V)$ are the data flow values
- $VarIn\left(s_{stop}\right) = \emptyset$ is the boundary condition, where s_{stop} is associated to a *Stop* command terminating a task
- $VarIn(s) = Ref(s) \cup (VarOut(s) \setminus Def(s))$ is the data flow value immediately before a basic block
- $VarOut(s) = \bigcup_{p \in Pred(s)} VarIn(p)$ is the data flow value immediately after a basic block

In Figure 6.3, the variable a is not live after definition (1) because it is killed by operator (2) before being used. Thus, it is **not** considered as significant for the system state before operator (2).

6.3.3 Reaching Definition Analysis and Constant Analysis

Reaching Definition Analysis provides information on where in a task each variable may have been defined when control reaches that position. This allows us to reason about whether a statement is constant and can safely be excluded from the state vector. Variables are not considered as relevant for the system state if they are reached only by constant definitions (e.g. $a = 5$) of the same value. Informally, a definition d reaches point p if there is a path from the point immediately following d to p, such that d is not

killed along the path. A definition d of a variable x is said to be *killed* if there is any other definition of x along the path. For this analysis, only the previously identified live variables are taken into account. By replacing an occurrence of a variable by its constant value, the variable may in turn not be live any more after the corresponding reaching definitions. Section 6.3.5 points out a strategy to overcome this circular dependency.

The transfer equations for Reaching Definition Analysis are defined as follows:

- $DefIn, DefOut \in \wp(S)$ are the data flow values
- $DefOut(s_{start}) = \emptyset$ is the boundary condition, where s_{start} is associated to the *Start* command at the beginning of a task
- $DefOut(s) = varFunc(Gen(s)) \cup (DefIn(s) \setminus varFunc(Kill(s)))$ is the data flow value immediately after a basic block
- $DefIn(s) = \bigcup_{p \in Pred(s)} DefOut(p)$ is the data flow value immediately before a basic block

In Figure 6.3 for example, the definition (7) is reached by itself (due to the loop) and operator (2). As both definitions are not constant, the analysis gets to the conclusion that the variable represents the system state after definition (7).

6.3.4 Local dependence analysis

Local Dependence Analysis reveals how control and data flow of a task are impacted by both non-deterministic decisions and incoming parameters of Events and Requests. The intuition behind this analysis is that several executions of a task are similar if a task is neither influenced by its environment nor it contains non-deterministic operators. At points where either of the two influences come into play execution traces are likely to converge or diverge. Input dependence analysis serves two goals: on the one hand it helps us to spot points where a system state should be saved and compared to previous states (referred to as check points, cf. section 6.4). On the other hand, dependent variables are likely to discriminate the task executions from previous runs and should thus be privileged when comparing system states.

6.3.4.1 Dependence Relations

One certainly has an intuitive understanding of the notion of dependence in the context of concurrent tasks. For instance, we are aware that a reference to a variable depends on its definition, the execution of a block may depend on a condition, in-deterministic decisions may alter the execution of a task, the filling level of a channel depends on the performed *Write* and *Read* operations and sending a parameterized event to another

task probably has an impact on its behavior. However, to clearly define dependence analysis, it is unavoidable to substantiate these straight forward relationships. The following identities have been established in accordance with [29] and [88] but have been adapted to particularities of the DIPLODOCUS profile.

- **Control dependency**: Let s_1, s_2 and s_3 be three basic blocks of a DIPLODOCUS task T, s_2 control depends on s_1 iff whether or not s_1 determines the execution of s_2. If s_2 control depends on s_1, and there does not exist a statement s_3, which is on the path from s_1 to s_2 and on which statement s_2 control depends, then s_2 is directly control dependent on s_1, denoted by $CND(s_2, s_1)$. For example, basic blocks enclosed in the body of *Loop* or *Choice* statements control depend on the closest basic block associated to a condition.

- **Data Dependency**: Let s_1, s_2 be two basic blocks of Task T, if variable $v \in Def(s_1)$ and $v \in Ref(s_2)$ and there is a path from s_1 to s_2 only comprising basic blocks s_p for which $v \notin Def(s_p)$ holds, then s_2 data depends on s_1 denoted by $DAD(s_2, s_1)$. For example, a basic block data depends on another basic block if a value computed in the latter has an influence on a value computed in the former.

- **Selection Dependency**: Let s_1 and s_2 be two basic blocks of a task T, s_2 selection depends on s_1 iff whether or not s_2 can be executed depends on a non deterministic selection made in s_1. Selection dependence is denoted by $SED(s_2, s_1)$. Significant non-deterministic DIPLODOCUS operators such as *Random* and *Random Choice* impact either control flow, data flow or both. Selection Dependency is thus established incrementally by propagating direct control and data dependency as described in section 6.3.4.

- **Synchronization Dependency**: If an inner state of a communication medium *comm* (Channel, Event, Request) depends on a basic block s_1, the communication medium synchronization depends on s_1, denoted by $SYD(comm, s_1)$. For example, a DIPLODOCUS Channel depends on basic blocks associated to blocking *Read* and *Write* operators.

- **Communication Dependency**: Let s_1, s_2 be basic blocks corresponding to a *Wait event* and a *Send Event* statement respectively, s_1 communication depends on s_2 if $ID(s_1) = ID(s_2) \wedge Comm(s_1) = Comm(s_2) \wedge Def(s_1) \neq \emptyset \wedge Ref(s_2) \neq \emptyset$. Communication dependency is denoted by $CMD(s_1, s_2)$. This dependency can be though of an inter task data dependency. Informally, it states that data dependency is also propagated via parameters of DIPLODOCUS events.

6.3.4.2 Dependence discovery algorithm

The following algorithm propagates dependencies on non-deterministic decisions by marking basic blocks with the $Dep(s)$ flag:

1. At first, the initial marking is established: for all basic blocks s corresponding to random statements (*Random*, *Random Choice*, *Select Event*) and to statements impacted from outside the task (*Notified*, *Wait Event* receiving parameters) $Dep(s) = true$ is set. The initial marking of *Wait Event* commands accounts for communication dependency. Following the definition of synchronization dependency, a potentially blocking synchronization command (*Read*, *Write*, *Send Event*, *Wait Event*) propagates its dependency to its related Channel, Event or Request.

2. Basic block s_2 is marked ($Dep\,(s_2) = true$) if it directly depends on basic block s_1 and basic block s_1 is already marked ($Dep\,(s_1) = true$), thus $Dep(s_2) = Dep(s_1) \wedge (CND\,(s_1, s_2) \vee DAD\,(s_1, s_2))$ with CND and DAD from section 6.3.1. This step is iterated until a steady state is reached and no further marking is produced.

In Figure 6.3, operator (2) is marked first as it receives two parameters a and b from the outside. Given that the *Choice* operator (3) is reached by the definition of a and b in (2), it is said to be data dependent on (2). Subsequently, the marking is extended to operators (3) to (8) due to their control dependency on operator (3).

6.3.5 Putting it all together

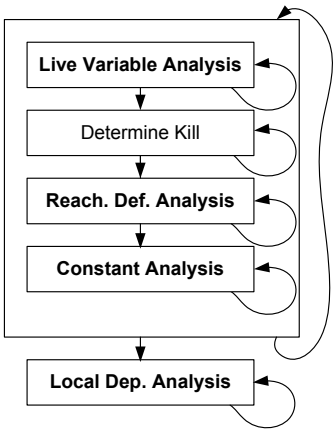


Figure 6.4: Cascaded Static Analysis

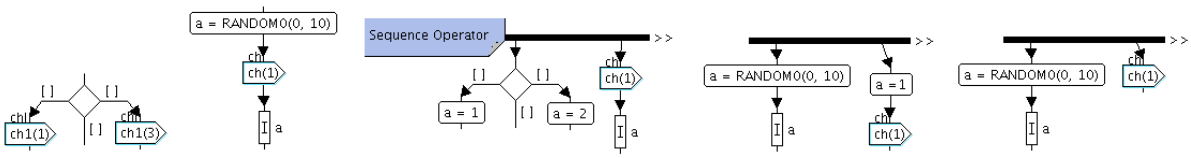
Figure 6.4 illustrates the interplay of the employed static analysis techniques. The loop back arrows indicate an iterative behavior of the algorithm, involving two nested loops. The outer loop terminates as soon as data flow values remain constant, while the inner loop iterates over all basic blocks of a task. Data flow equations defined in section 6.3.2 and 6.3.4 are solved in this fashion starting from the boundary condition and an initial estimate for data flow values. Live Variable Analysis first determines the set of significant variables at the entry and the exit of each basic block. Subsequent analysis

steps only consider basic blocks with variable definitions proven to be live. The next stage consists in identifying the definitions (e.g. basic blocks) that are killed by each basic block. This information is compulsory for Reaching Definition Analysis, which is carried out thereafter. The constant analysis aims to replace variable occurrences by constants whenever possible. The condition for variables to be replaced is easily expressed with the knowledge of reaching definitions: in order to replace the variable a, a basic block must only be reached by definitions for a referring to the same constant value. In so doing, the whole procedure is required to start over, as a removal of assignments and variables invalidates results of previous examinations. The first four stages are iterated (see Figure 6.4, big surrounding box) until the constant analysis does not detect new constant expressions any more. Intermediate data flow values have then converged to the final solution: $DefIn(s)$ to $blockFunc(In(s))$, $DefOut(s)$ to $blockFunc(Out(s))$, $VarIn(s)$ to $varFunc(In(s))$ and $VarOut(s)$ to $varFunc(Out(s))$.
When the aforementioned steady state is reached, the dependence analysis covered in section 6.3.4 concludes the overall procedure.

6.4 Checkpoint identification

The last section was concerned with the identification of significant parts of a state vector given a position in a task. It was argued that there are often elements of the state vector (such as task variables, Channels, Events and Requests) that do not influence the future behavior of the model. In that case, they can be safely disregarded in order to speed up the calculation of a state hash and to save memory for state dumps. Another important issue is to determine at which point in DIPLODOCUS tasks states should be saved and compared because control flow is likely to join. In the following, these particular points in a task are referred to as *checkpoints*. Checkpoint Analysis is conservative in the sense that even if we miss a check point, it only leads a small increase of simulation time. The simulation of two branches is carried on a bit further than necessary, which does not at all distort simulation results. Our assumption is that executions are only to be merged after inter task synchronization operators like *Send Event*, *Wait Event*, *Write* and *Read* operators. Moreover, we consider commands as being atomic, meaning that a command of length n does not require states to be compared n times but only once, after the execution of unit n. However, the state hash is updated after each transaction, as explained in Section 6.5.5.

A first attempt to elect check points would just consist in selecting all synchronization commands. But thanks to the previously performed static analysis of tasks, we can make educated guesses on where execution traces may be joined. The objective is to balance performance degradation due to state hashing on the one hand and due to unnecessarily following a branch of execution on the other hand. To this end, five cases have been identified that give rise to a checkpoint (cf. Figure 6.5). Therein, a synchronization operator is said to follow another command if there is a path of control flow connecting both which does not comprise another synchronization operator. To simplify matters,



(a) Task1 (b) Task2 (c) Task3 (d) Task4 (e) Task5

Figure 6.5: Example Tasks for Checkpoint Selection

we use a sloppy terminology in the following list. A command is assumed to directly expose the *Dep* (*s*) flag, thereby omitting the indirection via the dedicated basic block. As a basic block is mapped onto exactly one command, this should not cause any ambiguity. Checkpoints are defined to be synchronization commands . . .

1. . . . being associated to a channel, event or request having at least one operator where *Dep* (*s*) = *true* (in Figure 6.5a both commands are dependent).

2. . . . following a command that assigns to a variable a random value or a value generated outside the task (in Figure 6.5b)

3. . . . following a join of control flow if the synchronization operator is reached by more than one definition for the same variable, at least one of them having *Dep* (*s*) = *true* (Figure 6.5c depicts a *Write* command being reached by two dependent definitions of the variable a)

4. . . . following a command which kills at least one definition having *Dep* (*s*) = *true* (Figure 6.5d shows a random assignment being killed by the assignment $a = 1$, given that the variable is used later in the task)

5. . . . following a command where a variable goes out of scope which is reached by at least one definition *Dep* (*s*) = *true* (represented in Figure 6.5e if we assume the variable not to be used any more)

6.5 Implementation Issues

Static analysis of DIPLODOCUS tasks is accomplished in the context of the code generation stage. UML models are transformed into C++ code by means of a dedicated module integrated in TTool. This section covers some technical issues important for the realization of the aforementioned concepts, without focusing on the interplay with the TTool framework. Chapter 7 catches up on this by positioning code generation and static analysis in the overall tool chain.

6.5.1 Bit vector representation of dependencies

Data flow equations defined in section 6.3.2 and 6.3.3 are most efficiently represented in terms of bit vector operations. The semantics of a single bit is defined as the significance of a particular variable in case of Live Variable Analysis. For Reaching Definition Analysis, a bit denotes whether a basic block is reached by a particular definition. That means, Live Variable Analysis of a DIPLODOCUS task operates on bit vectors whose size amount to the sum of task variables, channels and events used by the task. For Reaching Definition Analysis, the bit vector must have as many elements as there are variable definitions in the task. *Action* commands make definitions explicit, but many other commands provide definitions for variables (*Wait Event*, *Random*, *Notified* commands, etc.). The union of sets is simply computed with the logical OR of bit vectors. The difference of two sets $S_1 \setminus S_2$ translates to the negation of vector T_2, which is combined with vector T_1 using a logical AND. For example, $Ref(s) \cup (VarOut(s) \setminus Def(s)) \Rightarrow Ref \vee (VarOut \wedge \neg Def)$, Ref, $VarOut$ and Def being bit vectors. This method avoids the maintenance of lists or other data structures to hold intermediate data flow values.

6.5.2 Propagating static analysis results to the simulator

The propagation of results of the static analysis to the simulation environment is accomplished with interface being implemented by descendants of a DIPLODOCUS command (*TMLCommand* class). The C++ code generator of TTool transforms each DIPLODOCUS task to a class, in the scope of which the commands of a task are instantiated. Commands of whatever type (descendants of *TMLCommand*, cf. Section 5.6.1) are initialized with a bit vector in the form of a string of variable length. In this string, each bit stands for a tasks variable, a channel or an event and denotes whether the latter is significant at the given point in the task. Commands leverage this information to constrain generation of state hashes to the relevant part of the state vector, and to (partially) avoid a regeneration of the hash if some components of the state vector remain unchanged.

6.5.3 The IndeterminismSource interface

The simulation kernel relies on an interface called *IndeterminismSource* to obtain information about the indeterministic behavior of a command. Random Choice, Random and Execi Interval commands implement both methods of the interface: *getRandomRange* and *setRandomValue*. The former returns the number of different executions permitted by the command, the latter explicitly selects one of them. For conventional simulation, a random generator picks the execution branch to be followed, whereas in exhaustive simulation all possibilities are covered iteratively. In the example given in Figure 6.6, a *getRandomRange* is sent (step 1) to three DIPLODOCUS commands: a Random command, a Choice command and a Execi Interval command. All commands will return the same value (step 2), namely 3 because they all subsume 3 possible executions. The second row of Figure 6.6 shows the equivalent DIPLODOCUS operator if the second

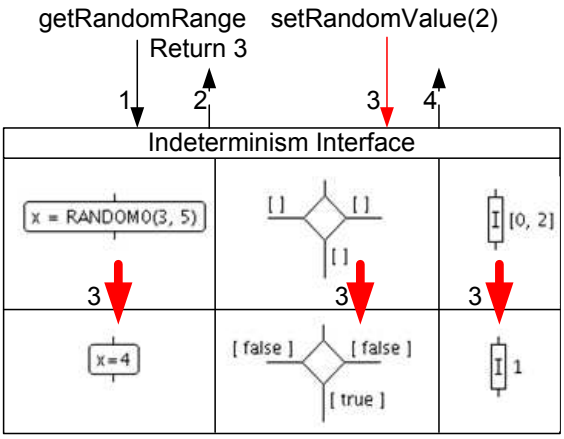


Figure 6.6: Illustration of the IndeterminismSource interface

execution is selected by calling *setRandomValue(2)* (step 3). The Random Command is transformed into an assignment of the second value within the given range, the guards of the choice command are disabled except the one of the second branch, and the Execi Range Command is resolved to a simple ExecI with the complexity of 1 unit (second value in the range).

6.5.4 Exhaustive and coverage driven Simulation

Within the simulation kernel (class *Simulator*), a dedicated procedure (*exploreTree*) directs the exploration mode. In accordance to the backtracking algorithm, it incrementally builds possible paths through the application model, and employs results obtained from the aforementioned static techniques to decide whether to abandon a candidate. The exploration mode yields a higher coverage of the application model as conventional simulation. Algorithm 1 presents the exhaustive coverage of the application model. At first, the model is simulated until one of three conditions is met:

1. a command with an indeterministic behavior is reached
2. the simulation terminates normally (thus all tasks either get blocked or terminate)
3. an already encountered state is reached.

In the two latter cases (handled by the if branch in Algorithm 1), the currently executed branch can be aborted. The detection of a known system state signals that an equivalent branch of execution has already been examined and so that pursuing the current branch will not provide new insights. If simulation has stopped due to an indeterministic command (else branch in Algorithm 1), the current simulation state is first saved to variable

s, and then a loop iterates over all possible valuations of the random variable (method *getRandomRange*, member of the *IndeterminismSource* interface). In the loop body, after the saved state s has been restored, the valuation to be explored is communicated to the indeterministic command (method *setRandomValue*, member of the *IndeterminismSource* interface). Thereafter, the elected branch is processed recursively in the same fashion (recursive call to *exploreTree* in Algorithm 1).

Algorithm 1 Coverage driven simulation algorithm

```
begin
  exploreTree
  while not (currCmd.isRandomCmd() or simTerminated() or knownStateReached) do
        simulate()
  od
  if simTerminated or
    knownStateReached then
                        quitBranch
                    else
                        s = saveState()
                        i = 1;
                        while i<=currCmd.getRandomRange() do
                            restoreState(s)
                            currCmd.setRandomValue(i)
                            exploreTree()
                            i = i + 1
                        od
  fi
end
```

6.5.5 State hashing

Figure 6.7 sheds light on how state hashing is accomplished in order to detect recurring simulation states. The scenario is initiated with a *prepare* message (Step 1 in Figure 6.7) sent to a command that generates transactions, in the scope of the *Update Phase*. The reader may refer to Section 5.6.1 and 5.6.2 for a detailed explanation of simulation phases and the interplay of components. The scenario is encountered if the command, referred to as *Checkpoint* in Figure 6.7, is terminated, i.e. if it has executed the whole amount of its virtual length. Moreover, the command must have been declared as a checkpoint during the Checkpoint Analysis stage (see Section 6.4). In Step 2, the checkpoint sends a *refreshStateHash* message to its task in order to invalidate the state hash of local task variables. Then, it passes control on to *sc* of the class *SimComp* (Step 3, *checkForRecurringSysState*

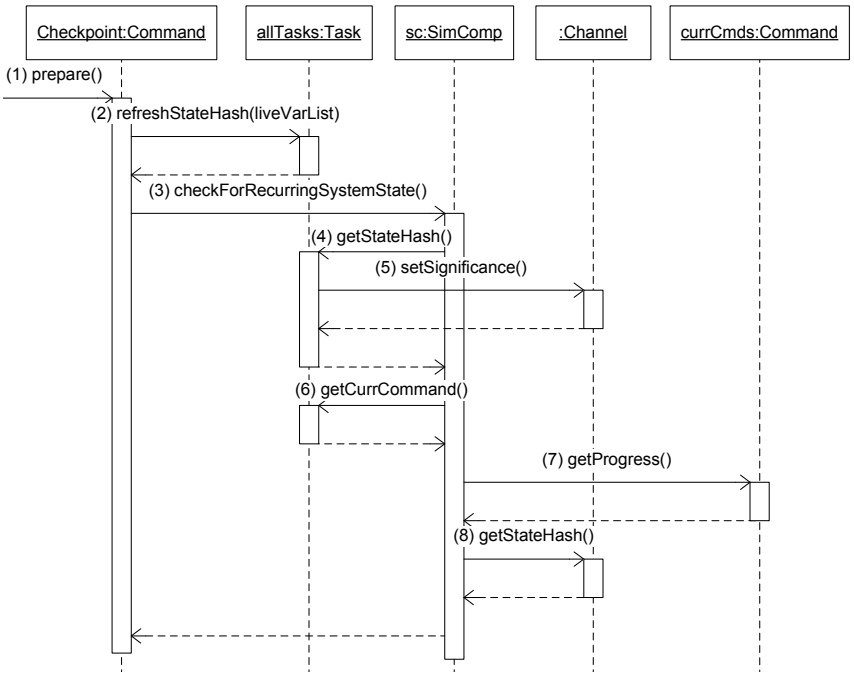


Figure 6.7: Sequence Diagram for state hashing during simulation

Operation	1 CPU	2 CPU
Send/Wait	1.24	1.19
Read/Write	0.77	0.72

Table 6.1: Simulator speed in MTrans./sec

message), which is in charge of managing all components relevant to simulation. In Figure 6.7, the unnamed components of type Task and Command (short for *TMLTask* and *TMLCommand*) stand exemplarily for all instances of these classes existing in the simulation environment. To trigger the recalculation of a task's state hash, *sc* issues the *getStateHash* message to all tasks. All tasks inform their channels, events and requests about their significance, that is whether a task will potentially perform operations on them in the future. This is achieved by means of the *setSignificance* method (Step 5). Thereafter, *sc* proceeds with the calculation of the hash by querying tasks for their current command (Step 6) and in turn the respective command for its current progress (Step 7). Finally, all channels, events and requests provide their contribution to the overall state hash when instructed with the *getStateHash* method.
To speed up the procedure, tasks, channels, events and requests locally calculate their hash is a lazy fashion. That means, that tasks, when receiving a *getStateHash* message, only recompute their hash if a progress was made since the last reception of *getStateHash*. Tasks only include variables in the state hash if they have proven to be significant during static analysis. This of course depends on the current position in the task.
In analogy to tasks, Channels, events and requests also refresh their hash only if they are significant, i.e. if they will potentially be accessed by any task in the future. Moreover, a state change since the last hash computation is also a necessary precondition for a revaluation of the hash. In case of events, all pending event occurrences have to be taken into account for the hash. If only event instances have been added, but not removed, the new hash is built incrementally based on the previous one. However, in case event instances were removed, the hash must be rebuilt from scratch. This inconvenience is inherent to the Jenkis hash function: elements added to the hash can be easily eliminated in a LIFO fashion, but for a FIFO semantics all add operations applied to the hash since the one of the element to be removed would have to be unrolled. We opted for the Jenkins Hash [56] because it does a good job in spreading similar keys, which are very common to DIPLODOCUS application states. This is due to the fact that a transaction only impacts a small subset of the state vector.

6.5.6 Experimental results

We carried out similar experiments to those presented in section 5.6.4 to determine the performance impact of state hashing. Again, we mapped an application comprising only Send/Wait commands and Read/Write commands onto two architectures composed of 1 and 2 CPUs respectively, a bus and a memory. We assumed 25% of the commands

to be checkpoints, where hash values of past states have to be compared with the hash value of the current state. In table 6.1 it can be seen that performance decreases by roughly 15% for event operations, and by roughly 10% for channel operations. This slight degradation should be more than compensated in case simulation branches can be merged.
The performance study was conducted on a 2.7 GHz host machine with 4 GB RAM.

6.6 From bits and pieces to model checking

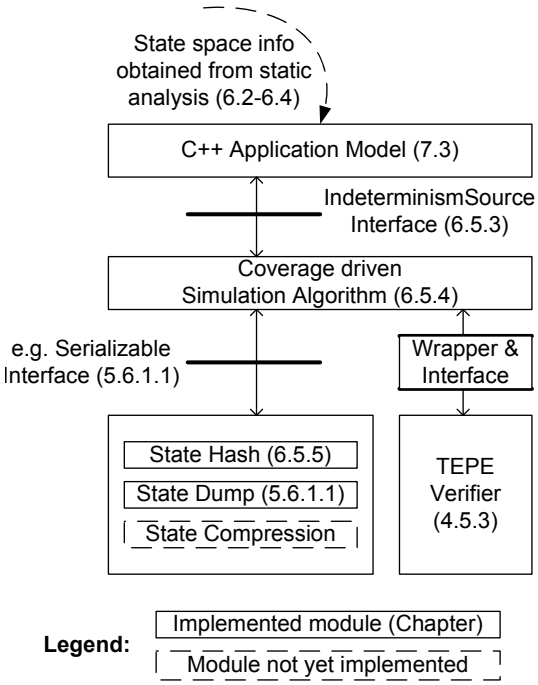


Figure 6.8: Leveraging presented techniques for model checking

Figure 6.8 shows how the techniques presented so far are combined to perform model checking of DIPLODOCUS models. The respective section of this thesis that elaborates on a module is indicated in parenthesis. Section 6.2-6.4 covered the generation of static information characterizing the significant part of the state vector and check points. This information is included in the C++ representation of the application, which is detailed in section 7.3. Together with the coverage enhanced simulation algorithm explained in section 6.5.4, we are able to exhaustively explore the state space of a DIPLODOCUS application model. The algorithm relies on the *IndeterminismSource* interface to query the application model for the range of random variables and also to communicate the valuation to be explored to the application model. The generation of state dumps (section 5.6.1.1) and state hashing (section 6.5.5) permits to compare encountered system

states with the current one and cancel simulation in case of a match. The simulation algorithm triggers state dumps by means of the *Serializable* interface (section 5.6.1.1). So far, state matching is accomplished with a hash and collisions in the hash table are not resolved. Future work should remedy this issue by introducing a state compression and comparison feature, see also section 9.3.3. The TEPE verifier (section 4.5.3) is supplied with traces in the form of signals and attribute values via a dedicated interface. The latter accomplishes a logical translation to keep simulator and verifier as independent as possible.The TEPE verifier finally determines the result of the injected TEPE properties.

6.7 Conclusions

In this chapter, we advocated the combination of static analysis and model checking techniques to enhance the simulation coverage of high level models of SoC. The methodology yields a trade-off between exhaustive and costly formal verification and efficient simulation exhibiting a limited coverage. That way, it complements the simulation strategy presented in Chapter 5 and the formal techniques already existing prior to this work. As opposed to conventional UML frameworks, both simulation and formal verification outreach the functional level by considering constraints imposed by hardware architectures

We firstly pointed out how the abstractions inherent to the model together with static analysis techniques can be leveraged to identify the relevant part of state vectors at given points in the model. Thanks to iteratively applied analysis stages, unused variables and constant expressions can be eliminated to speed up state comparison and memory consumption of state dumps.

Thereafter we defined *check points* as points in the model where different execution paths are likely to be joined thus permitting to abandon similar simulation runs. Local Dependence analysis allows for educated guesses on the positions of checkpoints, but even if some checkpoints are missed, simulation results are correct. The objective is to trade-off performance sacrificed for state hashing and for unnecessary reevaluation of control flow branches.

The next chapter positions the different functional blocks that have been discussed so far within the TTool development environment.

Chapter 7

Tooling

7.1 Introduction

So far, contributions of this work have been presented from a methodological perspective, mainly outlining the basic ideas and the underlying theory. Even if some insights into implementation issues were provided at the end of each chapter, it is still outstanding to put all these into the context of the toolchain. This chapter's objective is to close the circle and to review previously explained contributions in the overall framework of TTool, from a practical perspective. Section 7.2 elaborates on the internal structure of the elements of the toolchain, namely TTool and the simulation engine. Section 7.3 covers the transformation stage from the graphical UML model to C++ code. Interactive simulation features and their importance for debugging are showcased in Section 7.4. Thereby, special attention is drawn to the interaction between the simulator and TTool, which were developed in different programming languages.

7.2 Design Flow Revisited

Figure 7.1 depicts the toolchain that implements the contributions of this thesis, the latter being marked with a thick border. Third party tools are denoted with rectangles with rounded corners. TTool is based on the widespread model view controller design pattern. At its topmost level, diagramming facilities permit developers to draw class diagrams, activity diagrams and mapping diagrams. As we do not involve any third party tool at this stage, bugs can be easily corrected without relying on official channels and diagram animation features can be closely integrated. The Controller layer interprets modifications on the graphical model so as to construct a more abstract model which is independent of graphical elements. This abstract model solely reflects the semantics of DIPLODOCUS and is the starting point for transformation either to formal or to simulation models. Formal code generation and integration of 3rd party formal tools (CADP, UPPAAL) were already existing prior to this work [16] and are therefore not further discussed. This work addresses the generation of C++ code by means of a ded-

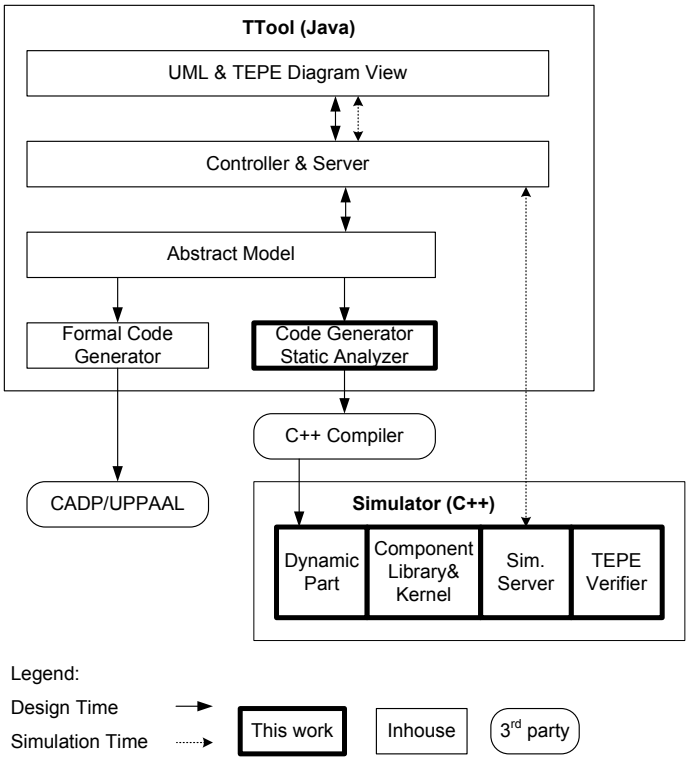


Figure 7.1: TTool toolchain

icated module integrated into TTool. This C++ code generator produces one separate class file per DIPLODOCUS task along with a file which instantiates application and architecture components. Moreover, it establishes links between the latter according to the mapping model. As shown in chapter 4, TEPE diagrams have their counterpart as a C++ class as well. We now reach the boundaries of the TTool application: generated C++ classes are compiled with the GNU C++ compiler and linked with the static part of the simulator. As depicted in Figure 7.1, the whole simulator is an achievement of this thesis. Architecture components for instance do not need to be generated dynamically, a parameterization is sufficient to conform the C++ model to the UML model. Simulation kernel, simulation server and TEPE verifier are static parts as well. It should be emphasized that model transformation for simulation is not a one way road: simulation results are back propagated to the original UML model, so that the user does not have to be aware of the executable model. This is accomplished at simulation run time, as the dashed arrows with the small arrowhead suggest. From a functional perspective, the enhancement of the environment brought the following improvements with respect to shortcoming of related work summarized in section 3.7:

- The lack of simulation engines exploiting properties of high level models lead us to an inhouse development of a simulator. It offers an efficient way to simulate DIPLODOCUS models, thus complementing existing formal verification facilities of the environment.
- DIPLODOCUS differs from other modeling methodologies in that data dependent decisions are made explicit by means of indeterministic operators. This property together with DIPLODOCUS' ability to abstract but still to account for control flow alternatives is leveraged for an intermediate application coverage in-between conventional simulation and exhaustive exploration.
- A rupture in the design flow is prevented which arises when UML models are verified with obscure logical formulas. The TEPE language seamlessly integrates the verification stage into a full-UML environment.
- As opposed to state-of the art UML simulators, our engine is not restricted to the mere functional domain but considers the impact of shared hardware resources. Performance figures and compliance to functional properties are obtained on the fly, at simulation time.
- Interactive simulation remedies the shortcoming of other simulators to generate executable models which are decoupled from their UML counterpart. The DIPLODOCUS environment provides direct feedback of simulation results to the UML model. UML models may be animated to illustrate simulation progress.

The next section elaborates on the connector between TTool and the simulator, namely the code generator.

7.3 Automated model transformation

To get an idea of how a UML model translates into C++ code, two of the three file types built by the code generator are exemplified. The first file type is generated once for a DIPLODOCUS model, and saved to a file named *appmodel.cpp*. An excerpt of this file is given in Listing 7.1. The contained code instantiates all components needed for simulation and adds them to internal lists using methods prefixed with *add*. First of all, in lines 4-6 CPUs and their schedulers are created. Even if only two schedulers (Round Robin and Priority based) have been defined so far, complex configurations can be achieved with a hierarchical composition of schedulers. The simulator could also effortlessly be enhanced with other scheduling algorithms. In lines 8-9/11-12, a bus and a memory are instantiated. CPUs may be connected to several buses and buses in turn may subsume several independent communication channels (not to confuse with DIPLODOCUS channels). This complexity is managed by *BusMasters*, added to the simulation environment in lines 14-16. Channels are initialized (lines 18-21) with references to the communication components they are mapped onto. For example, the *newSamples* channel is mapped onto the Crossbar and the *DDR2* memory. Events (lines 23-25) and requests (lines 27-29) are not mapped onto the communication infrastructure and are therefore just initialized with zeros and the FIFO size. The same schedulers that are used with CPUs are suitable for buses. In lines 31-32, a new priority based scheduler is assigned to a bus called *LeonLocalBus*. Finally, tasks (lines 34-39) are instantiated with references to their associated CPUs and all channels, events and requests they are connected to. In case a tasks runs on a multi core CPU, it receives a reference to every single core.

Listing 7.1: Main model file appmodel.cpp

```
1  class CurrentComponents: public SimComponents{
   public:
   CurrentComponents():SimComponents(498536312){
     PrioScheduler* Leon_scheduler = new PrioScheduler("Leon_PrioSched",0);
     CPU* Leon0 = new SingleCoreCPU(7, "Leon_0", Leon_scheduler, 1, 1, 1, 5, 20, 2, 10, 10, 4);
6    addCPU(Leon0);
     ...
     Bus* LeonLocalBus_0 = new Bus(5,"LeonLocalBus_0",0, 100, 4, 1,false);
     addBus(LeonLocalBus_0);
     ...
11   Memory* DDR2 = new Memory(3,"DDR2", 1, 4);
     addMem(DDR2);
     ...
     BusMaster* Leon0_LeonLocalBus_Master = new BusMaster("Leon0_LeonLocalBus_Master", 0,
     1, array(1,(SchedulableCommDevice*)LeonLocalBus_0));
16   Leon0->addBusMaster(Leon0_LeonLocalBus_Master);
     ...
     TMLbrbwChannel* channel__Rec80211p__newSamples = new TMLbrbwChannel (52,
     "Rec80211p__newSamples", 4, 2, array(2,FEP0_Crossbar_Master, Deinterleaver0_Crossbar_Master),
     array(2, static_cast<Slave*>(DDR2), static_cast<Slave*>(DDR2)), 8, 0, 0);
21   addChannel(channel__Rec80211p__newSamples);
     ...
     TMLEventBChannel* event__Rec80211p__consumeOK1 = new TMLEventBChannel(67,
     "Rec80211p__consumeOK1", 0, 0, 0, 0, 0);
     addEvent(event__Rec80211p__consumeOK1);
26   ...
```

```
    TMLEventBChannel* reqChannel_Rec80211p__ChannelEstimation = new
    TMLEventBChannel(81,"reqChannelRec80211p__ChannelEstimation", 0, 0, 0, 0, 0, true);
    addRequest(reqChannel_Rec80211p__ChannelEstimation);
    ...
    LeonLocalBus_0->setScheduler((WorkloadSource*) new PrioScheduler("LeonLocalBus_PrioSched",
    0, array(1, (WorkloadSource*)Leon0_LeonLocalBus_Master), 1));
    ...
    Rec80211p__PacketDispatcher* task__Rec80211p__PacketDispatcher = new
    Rec80211p__PacketDispatcher(11,0,"Rec80211p__PacketDispatcher", array(1,CPU00),1
    ,event__Rec80211p__consumeOK1
    ,reqChannel_Rec80211p__PacketDispatcher
    );
    addTask(task__Rec80211p__PacketDispatcher);
  }
};
```

For each DIPLODOCUS task a corresponding C++ class is written to a separate file. A shortened version of such a class file is shown in Listing 7.2. A class file is of course accompanied by a trivial header file which merely contains the declarations of functions exemplified in the Listing. As stated before, pointers to all communication primitives are passed to the task (lines 1-6). In lines 7-8, local task variables are assigned their initial value. The semantics of DIPLODOCUS operators is captured by separate classes which are descendants of *TMLCommand* (compare section 5.6.1). Instances of these classes are constructed in lines 9-13. Constructors of commands are invoked with a binary coded list of significant state variables in the form of a string. This information is vital to efficient state hashing (cf. section 6.5.5). Moreover, function pointers are passed to command constructors as it will be explained later. Lines 14-15 relate two communicating tasks with their channel, event or request. Lines 17-20 chain the commands according to the control flow stipulated in the UML model.
In addition to numeric parameters, constructors of commands receive function pointers. This is to avoid separate class definitions for every occurrence of a DIPLODOCUS operators. For example, *Action* commands perform user defined assignments, *Wait* or *Send* command may transfer values between the FIFO of an event and task variables, and the complexity of *Execi* commands (virtual length) may be given in the form of a complex equation. Furthermore, these operations often require access to local task variables. To account for this, three types of task member functions have been defined:

- Functions transferring values between parameters and task variables: an example can be found in lines 24-26, where a task variable *datalen* is assigned the value of the first parameter of an event.
- Functions computing the virtual length of DIPLODOCUS operators: in lines 39-40, the length of an *Execi* operator is just given by a single variable, called *datalen*.
- Functions performing the action specified in an *Action* operator: lines 35-36 show an example of a variable assignment.
- Functions determining the number of active guards of a *Choice* operator: the function in lines 29-32 results from a *Choice* operator with two guards:
(1) $datalen > 0$ represented by the first if block and (2) *else* leading to the second if

block. The return value is a randomly chosen candidate of all active guards. If no guard is applicable, the *else* guard is taken if existing.

As stated in Chapter 5, *readObject* and *writeObject* methods rebuild a simulation component from a byte stream or generate a byte stream from a simulation component respectively. Together with the *reset* method, they implement the *Serializable* interface which permits to save and restore simulation states. The *getStateHash* method (lines 56-61) adds task variables to the state hash in case they are considered as significant. Furthermore, all channels, events and requests which will potentially be accessed in the future are marked as significant (*setSignificance* method).

Listing 7.2: Task model

```
Rec80211p__DATA_FEP::Rec80211p__DATA_FEP(ID iID, Priority iPriority
, std::string iName, CPU** iCPUs, unsigned int iNumOfCPUs
, TMLEventChannel* event__Rec80211p__consumeOK4
, TMLEventBChannel* request__Rec80211p__startSync2
, TMLEventBChannel* requestChannel
):TMLTask(iID, iPriority, iName, iCPUs, iNumOfCPUs)
,datalen(0)
,arg1__req(0)
,_wait209(209,this,event__Rec80211p__consumeOK4,(ParamFuncPointer)
 &Rec80211p__DATA_FEP::_wait209_func, "\xf\x0\x0\x0", true)
,_lpChoice208(208, this, (RangeFuncPointer)&Rec80211p__DATA_FEP::_lpChoice208_func, 2, 0, false)
,_execi205(205,this,(LengthFuncPointer)&DIPLODOCUSDesign__DATA_FEP::_execi205_func,0,1,
 "\xf\x0\x0\x0",false)
,_action262(262, this, (ActionFuncPointer)&Rec80211p__DATA_FEP::_action262_func, 0, false){
  event__Rec80211p__consumeOK4->setBlockedReadTask(this);
  event__Rec80211p__sendFEPCmd4->setBlockedWriteTask(this);
  ...
  _wait209.setNextCommand(array(1, (TMLCommand*)&_send212));
  _lpChoice208.setNextCommand(array(2, (TMLCommand*)&_request210, (TMLCommand*)&_request205));
  _action262.setNextCommand(array(1, (TMLCommand*)&_lpChoice208));
  _waitOnRequest.setNextCommand(array(1, (TMLCommand*)&_action262));
  ...
}

Parameter<ParamType>* Rec80211p__DATA_FEP::_wait209_func(Parameter<ParamType>* ioParam){
  ioParam->getP(&datalen);
  return 0;
}

unsigned int Rec80211p__DATA_FEP::_lpChoice208_func(ParamType& oMin, ParamType& oMax){
  if ( datalen>0 ){...}
  if (oMax==0){...}
  return getEnabledBranchNo(myrand(1, oC), oMax);
}

void Rec80211p__DATA_FEP::_action262_func(){
  datalen=1;
}

TMLLength DIPLODOCUSDesign__DATA_FEP::_execi205_func(){
  return (TMLLength)(datalen);
}

std::istream& Rec80211p__DATA_FEP::readObject(std::istream& i_stream_var){
  READ_STREAM(i_stream_var, datalen);
  TMLTask::readObject(i_stream_var);
  return i_stream_var;
}
```

```
49
   std::ostream& Rec80211p__DATA_FEP::writeObject(std::ostream& i_stream_var){...}

   void Rec80211p__DATA_FEP::reset(){
     TMLTask::reset();
54   datalen=0;
   }

   HashValueType Rec80211p__DATA_FEP::getStateHash(){
     if(_hashInvalidated){
59     _hashInvalidated=false;
       ...
       if ((_liveVarList[0] & 1)!=0) _stateHash.addValue(datalen);
       _channels[0]->setSignificance(this, ((_liveVarList[0] & 2)!=0));
       ...
64     }
     }
     return _stateHash.getHash();
   }
```

7.4 Interactive Simulation

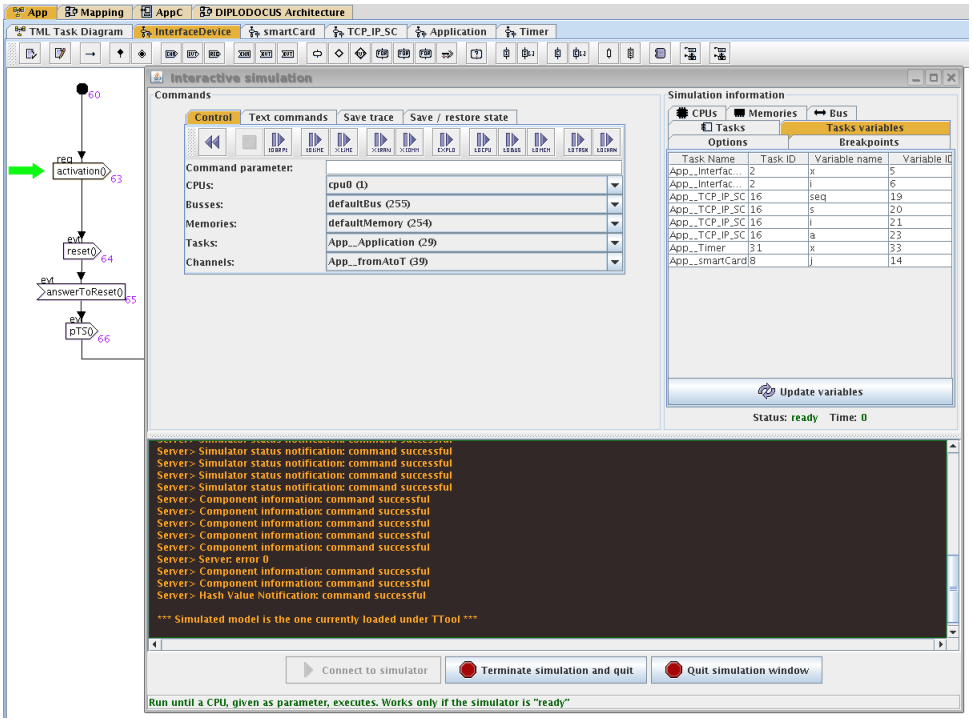


Figure 7.2: The Interactive Simulation Window

The simulation environment features an interactive exploration of an application mapped onto a particular architecture. After having developed the static view of the application in terms of classes, the behavioral view, the architecture and the mapping, the developer

first checks the syntax of the models. If the models comply to the DIPLODOCUS meta-model, the designer proceeds with the automatic generation of a C++ model. Once the sources have been compiled, the interactive simulation module is launched. All of the aforementioned stages are accomplished at the push of a button. No expertise in C++ programming, simulation or formal verification is required. The starting point for an interactive exploration is the window depicted in Figure 7.2. The interface provides the following simulation commands:

- Different flavors of run commands: a given amount of transactions, commands or time units can be simulated ...
- ... likewise the simulation may be interrupted when a particular hardware element (CPU, bus, bridge, memory) or an application entity (channel, event, request) processes a transaction.
- Reset the simulation to the initial state.
- Save and restore the simulation state, especially useful when several branches of control flow are to be looked into manually.
- Simulation traces may be provided in several formats: the text based format is a simple listing of all transactions encountered on a hardware component This format enables the automatic evaluation of traces and the interchange of data with other applications. The VCD format is supported for the sake of compatibility with standard waveform viewers. The VCD output basically captures bus states (read, write, idle), task states (ready, running, blocked, terminated), and CPU states (executing, idle, sleep mode). For debugging purposes of small designs, simulation results are displayed in the user friendly form of a Gantt diagram (depicted in Figure 7.4). Transactions are represented on a time line for each hardware component and colored according to their task of origin. Figure 7.3 visualizes the result of a coverage enhanced simulation in the form of a reachability graph.
- Breakpoints can be set graphically on any command within the UML activity diagram simply by selecting a dedicated option in the context menu. Two kinds of breakpoints are supported: conditional and unconditional breakpoints. Unconditional breakpoints stop the simulation whenever a specific command of an activity diagram is reached. Conditional breakpoints interrupt the simulation as soon as a condition (a function of task variables) is met.
- Coverage enhanced exploration is started with a dedicated button.
- Some commands return information on the simulation state and on the state of application and hardware components. Normally, the user does not explicitly use these commands as they merely serve as a means to propagate feed-back to the graphical user interface.

- Commands may also be supplied directly in text format, without employing the graphical interface. Thus, scripts can be written to automate the simulation steps.

TTool encompasses a graphical interface to direct the simulation (shown in Figure 7.2) and thus unburdens the user from familiarizing with simulation commands. The feedback from the simulation engine is exploited by the graphical user interface and is used to animate UML application diagrams. For instance, the current command of a task is highlighted so that the user is able to closely follow the simulation progress.
In addition to traces, simulation results comprise performance figures (cf. Figure 7.5) like the utilization of hardware elements, the contention delay for bus masters, the execution time of tasks, the average time a task gets blocked due to CPU contention, etc.
As a simple example, let us consider an algorithm having two main branches which significantly differ in terms of execution time and resource usage in general. For the performance evaluation of a specific architecture, it would be crucial to try out both alternatives. As a first step, the designer could benefit from the various conditional run commands so as to get a more intuitive view of the behavior of the application and the interaction of hardware components. The next step could be to reset the simulation and to set a breakpoint on the branch command which is crucial for the continuation of the simulation. The simulation will stop at the previously defined choice command therefore allowing the user to specify the branch to be explored. In combination with the feature of capturing simulation states, complex scenarios can be evaluated and meaningful traces be recorded. In our example, the user would certainly save the simulation state when reaching the choice command so that it can be restored to study other alternative executions. However, if interactivity is not desired, the exploration mode may automatically examine several executions.

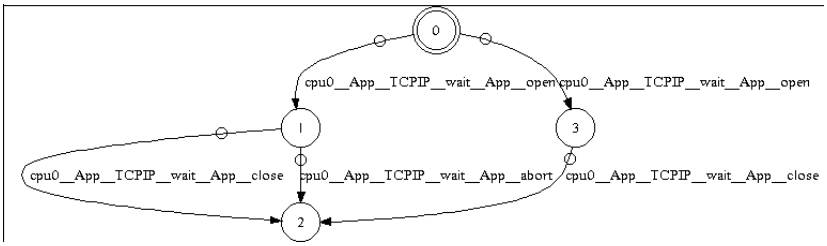


Figure 7.3: Simulation results in the form of a reachability graph

7.5 Frontend-Backend Communication

As illustrated in Figure 7.6, the simulator and the graphical user interface embedded in TTool are hosted in different processes communicating via a TCP connection. Therefore, the simulator and the graphical user interface can be run on different machines to increase performance. To get a better understanding of this interaction, let us now

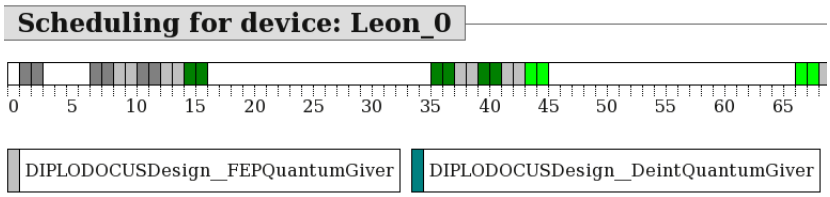


Figure 7.4: Simulation results in Gantt diagram format

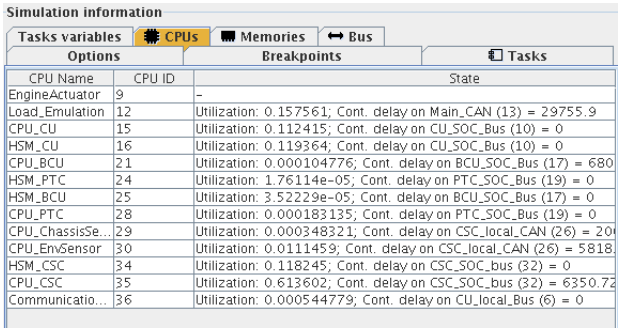


Figure 7.5: Tabulated benchmarks obtained from simulation

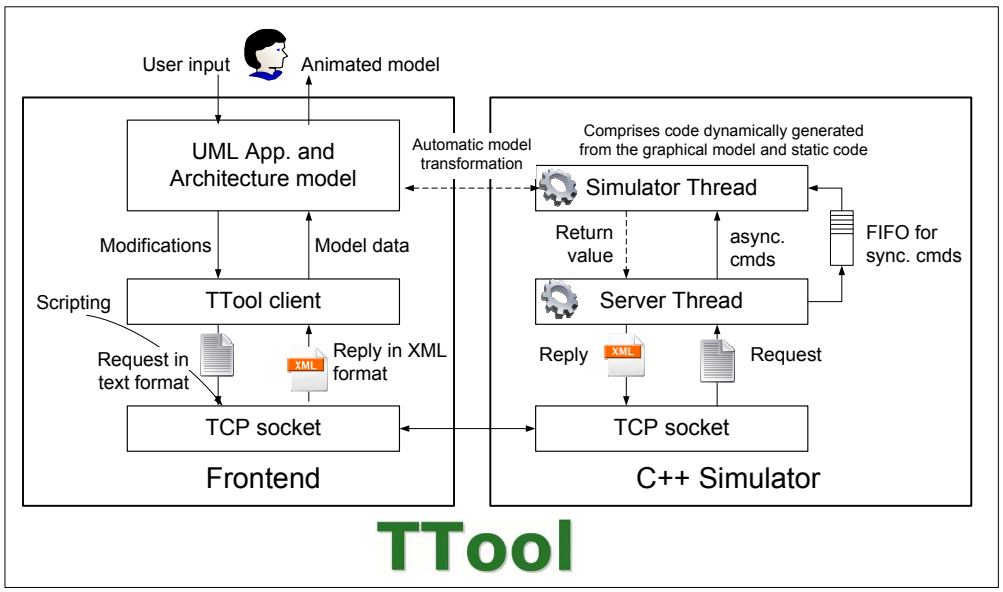


Figure 7.6: Interaction of the Frontend and the Simulator within the TTool Framework

follow a user request which aims at setting a breakpoint. The user selects the option by clicking on the respective command within the UML activity diagram. In turn, the logic of the graphical user interface identifies the concerned command and signals a modification to the TTool client. The latter may perform additional checks and wraps information about the command (its ID, ...) and the request into a message in text format. The message is sent over the network and received by the server thread of the simulator. The latter distinguishes so called synchronous and asynchronous requests. Asynchronous requests may be issued at any time and normally request information about the simulation without altering the simulation state. Asynchronous requests are handled in the scope of the server thread. Synchronous requests however directly impact the simulation state and must therefore be processed in order. Our breakpoint request is considered as such. Synchronous commands are carried out by the simulator thread which reads the FIFO entries one after another. In case of the breakpoint request, the simulator updates internal data structures accordingly and notifies the successful breakpoint insertion to the server. The server in turn encapsulates the reply into an XML message and sends it over the network. The TTool client subsequently interprets the message and informs the graphical user interface. The latter provides a feedback to the user indicating that the breakpoint has been set successfully.

7.6 Conclusions

This Chapter reviewed the contributions of this thesis from the implementation point of view. Emphasis was put on the practical part of this work, which aimed at proving the feasibility of aforementioned theoretical aspects. The initial configuration of the toolchain, prior to this work, included the graphical front end and transformation facilities to formal languages. The latter have been complemented with an efficient simulation framework, incorporating the contributions outlined in previous chapters: support of TEPE, an optimized simulation strategy for high level models and a variable coverage of models. Even after compilation of the executable model, the semantic link to the original model is not lost. Simulation progress is directly visualized within UML application diagrams and performance figures within architecture diagrams. Thus, the designer is given a powerful toolbox which is especially suited for performing early System Level Design Space Exploration. It may considerably alleviate the burden of experimenting with several different architectures so as to obtain performance key figures and to check compliance to functional properties. The following chapter will confront the tool with a realistic case study to prove its practical applicability.

Chapter 8

Evaluation

8.1 Introduction

So far, we have proposed the property specification language TEPE, a simulation framework for high level models of Systems-On-Chip, a methodology to enhance the coverage of latter and finally a tool suite that implements the whole design flow. Our efforts were motivated by shortcomings of existing environments (identified in chapter 3) with respect to abstraction level, simulation speed, representation of control flow and indeterminism. While evidence for the applicability of several concepts has been provided throughout this thesis, this chapter emphasizes on a more comprehensive example. It stems from the telecommunication domain and highlights the model of a receiver for the 802.11p standard, its properties as well as its simulation and verification. More precisely, the objective of this chapter is to

- exemplify the DIPLODOCUS modeling concept and its outcomes with a concrete existing system
- demonstrate that TEPE is capable to express relevant properties of SoC
- illustrate how outcomes of simulation can be used
- provide evidence for the usefulness of coverage enhanced simulation
- illustrate the interplay of different tool components.

However, confronting simulation results with performance measurements on the real hardware is out of scope of this work. With respect to non-functional properties, the comparison published in [55] make us confident that the DIPLODOCUS methodology yields insightful performance figures. The designer is indeed assisted in his effort to spot an implementation complying to the given (non-) functional constraints. Concerning functional properties, a refinement of DIPLODOCUS models preserving the latter is currently investigated in the scope of an ongoing dissertation [84].
This chapter is structured as follows: section 8.2 surveys the application domain and

the system to be modeled and defines related notions. Section 8.2.2 covers DIPLODOCUS models of application and architecture which will be leveraged to determine performance figures and compliance to properties in section 8.3. The latter section also analyzes the performance of the simulator, both in simulation and exploration (coverage enhanced) mode. Section 8.4 finally concludes this chapter.

8.2 Case study: An 802.11p receiver

Wireless communication systems have witnessed a rapid evolution over the past decades. The latest products support different radio spectra, protocols and access technologies and add new features while merging existing applications in only one device. This evolution triggers the need for flexible hardware platforms that are capable to deal with a broad variety of standards while optimizing silicon area and power consumption.
This is where the concept of **Software Defined Radio** (SDR) [83] comes into play. Its aim is to configure the behavior of radio devices through programming, rather than directly modifying a dedicated circuit. The advent of SDR bears resemblance with the evolution of calculators that culminated in the invention of programmable computers, whose instructions are not hard wired but stored in a memory. The strengths of SDR are for example the effective use of the spectrum, seamless mobility, maintenance cost reduction in networks and a faster deployment of new or modified standards. A challenge to be met is that radio systems fall into the category of real-time embedded systems and often need to be portable. Therefore time constraints, power consumption and silicon area are just as crucial as the functional correctness of the final product.
In [86], a joint ongoing research project of Eurecom and Telecom ParisTech is presented. The paper advocates the so called **ExpressMIMO-Card** (EC) which adheres to the SDR paradigm. EC incorporates generic signal processing blocks to be dynamically configured for standards such as 2G, 3G, 4G/LTE and members of the 802.11 protocol family (W-Lans). The standard modeled in this case study (802.11p) is destined for wireless communication in vehicular environments. Its purpose is to permit data exchange between two vehicles and between vehicles and the roadside infrastructure for applications such as toll collection, safety services, and commerce transactions.

8.2.1 The Eurecom ExpressMIMO-Card

The architecture of the ExpressMIMO-Card (EC) depicted in Figure 8.1 hosts the digital part of the physical layer (PHY). The provided functionality belongs to the realm of baseband processing, which is accomplished directly after analog to digital conversion (called Radio Front-end in Figure 8.1). Obviously, the architecture is structured in two parts: a module, named Processing or DSP Engine, implementing independent signal processing blocks as well as a control module orchestrating their interplay.
The **Processing engine** embraces a set of IP blocks dedicated to a particular type of signal processing. A crossbar permits the blocks to communicate independently of each other.

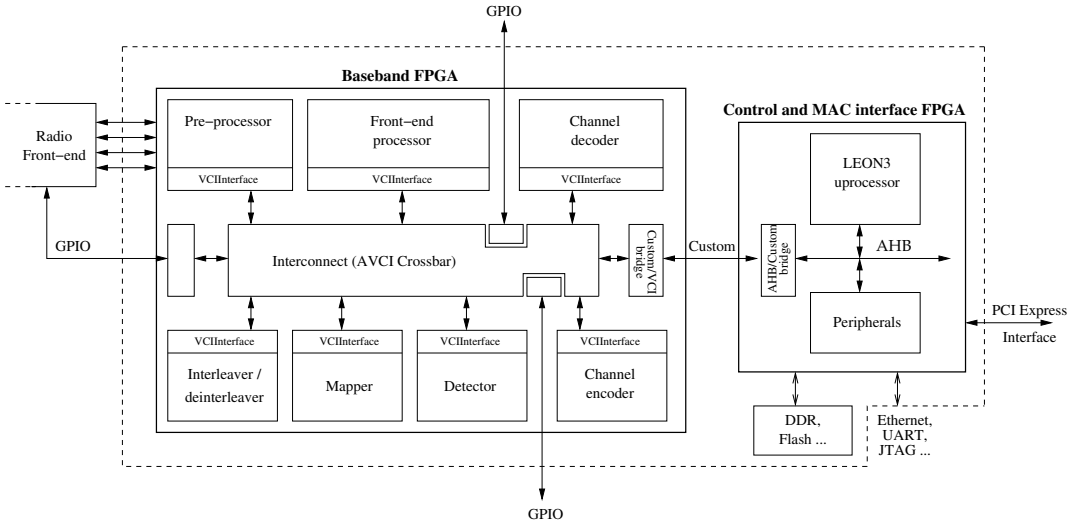


Figure 8.1: Baseband processing architectural overview

That way, the level of concurrency is only limited by the number of distinct destination blocks; the crossbar itself is non-blocking. To satisfy the SDR principle, blocks are highly configurable. Each block comprises a local memory, a data processing unit (IP core) and a DMA engine to import or export data from/to other blocks.
The **Pre-processor** serves as an interface to A/D and D/A converters and is responsible for sample rate conversion [111], sample rate adjustment and carrier frequency adjustment amongst others. **Mapper** and **Detector** modules are not relevant for the case study and are therefore not further discussed. The **Deinterleaver** is in charge of restablishing the original order of data samples that have been scrambled by the sender. The **Front-end-Processor** (FEP) is the basic toolbox for signal processing operations such as channel estimation, energy detection, and fourier transforms. The **Channel Decoder** attempts to recover the originally sent bit sequence even in the presence of transmission errors due to the wireless channel. It implements decoding standards for convolutional and turbo codes.
The control module is equipped with a SPARC CPU, peripherals, external memories and interfaces to a host PC. Its main responsibility is to program and configure IP blocks of the DSP engine and to initiate data transfers on DMAs. The synchronization between the control module and the DSP engine is accomplished by a set of interrupts signaling the end of data transfers and processing. Processing linked to layers above the physical one (such as MAC) may be executed on the PC. However, in the case study the board is considered to function in stand alone mode, making the interface irrelevant.

8.2.2 DIPLODOCUS model

This section elaborates on the DIPLODOCUS model of the decoder described previously. Its structure reflects the stages of the DIPLODOCUS methodology: identifying

IP/Module	Functionality Task	Data Transfer Task
Pre-processor	PacketGenerator PacketDispatcher	
Front-end Processor	FEP_Func	DMA_FEP_Func
Channel Decoder	ChDec_Func	DMA_ChDec_Func
Deinterleaver	Deint_Func	DMA_Deint_Func
Control module	Synchronization ChannelEstimation SignalFieldDetection DATA_FEP DATA_Deint DATA_ChDec	

Table 8.1: Mapping of IPs onto DIPLODOCUS tasks

functional entities by establishing a clear separation between functionality and platform, expressing communication between tasks in terms of DIPLODOCUS primitives, associating a behavior to each task and finally modeling the architecture and mapping application tasks onto it.

8.2.2.1 Identification of functional entities

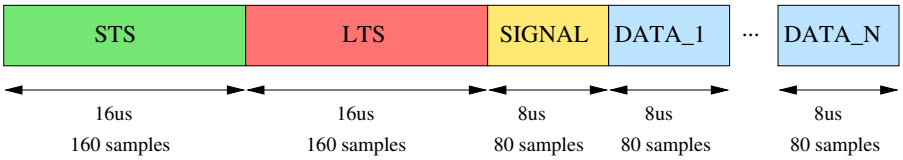


Figure 8.2: 802.11p packet

The first stage consists in consulting the specification and drawing clear boundaries between pieces of information concerning the application and the architecture. For instance, the notion of "IP" refers to both a behavior and an underlying execution hardware. To get a first intuition on the behavior IPs may exhibit, we consider the structure of an 802.11p packet shown in Figure 8.2. It is composed of a fixed size part and a variable size payload part. The former can be subdivided into a

- Short Training Symbol (STS), containing 10 repetitions of a known sequence to detect the beginning of a packet
- Long Training Symbol (LTS), containing guard intervals and two repetitions of the same known sequence for calculating a channel estimate

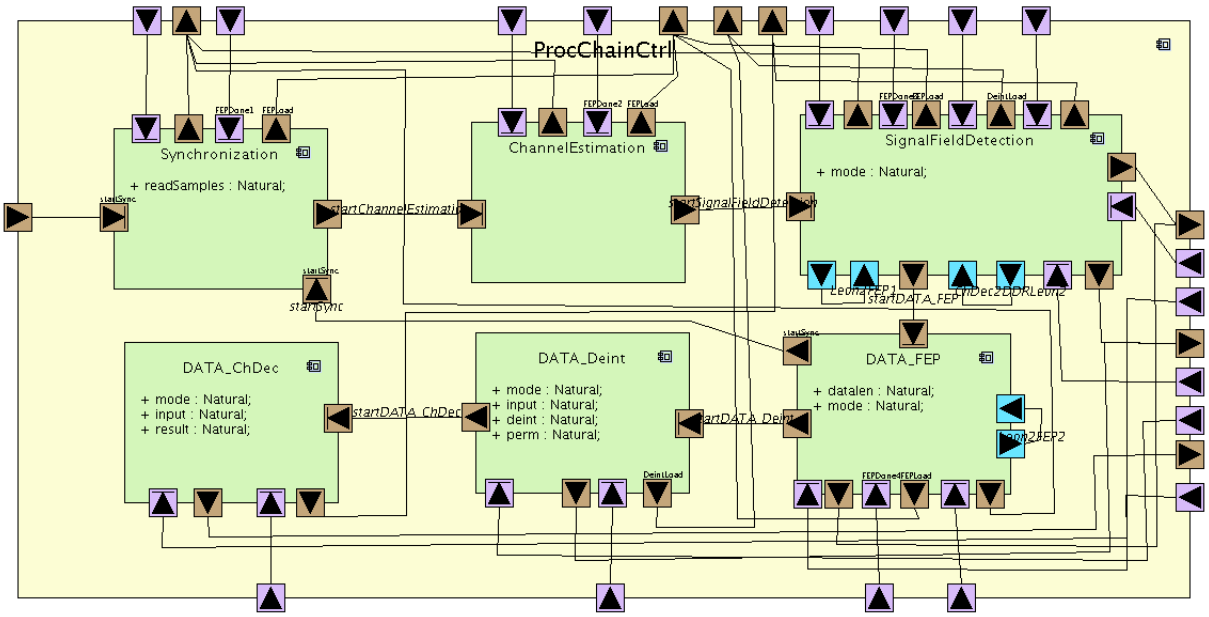


Figure 8.3: Excerpt from the DIPLODOCUS application model of the 802.11p receiver

- SIGNAL, containing meta-data that specifies the length and the decoding scheme of the subsequent DATA field.

The DATA field may contain between 1 and 1366 DATA symbols a 80 samples and constitutes the actual payload to be transmitted.
It is out of scope of this work to discuss the modeled processing steps and their mathematical background in detail. However, sufficiently many details[1] are provided to convince the reader that the receiver has been adequately represented.
In DIPLODOCUS, the first challenge to be met is to abstract data, as the methodology does not allow for real data samples to be manipulated. As stated in chapter 2, this restriction comes with the advantage of limiting the state space. In turn, this makes models amenable to exhaustive simulation (cf. chapter 6) and formal methods. In signal processing applications, the complexity of filter and transformation routines is usually not data dependent. However, for our receiver, the workload imposed on the architecture strongly depends on

1. the length n of the payload of each packet (named DATA_x in Figure 8.2, $x = 1 \ldots n$),
2. the inter packet arrival time, and
3. the respective modulation used to transmit the payload (such as BPSK, QPSK and QAM).

[1] so just lagom many

A common practice in DIPLODOCUS is to externalize data dependent decisions in a dedicated task, a so called test bench. Random variables representing the decisions are passed to all tasks in the processing chain that would be data dependent on lower abstraction levels. That way, a consistent behavior according to a specific class of input data is assured across several DIPLODOCUS tasks.
Table 8.1 establishes the correspondence between the previously presented IPs and DIPLODOCUS tasks. As DIPLODOCUS tasks are purely functional entities, this step implies a separation of the IP's architecture from its functionality. We will first discuss the model from a functional perspective, architecture issues are deferred to the next section. Table 8.1 discriminates two categories of tasks, namely Functionality and Data Transfer Tasks. Functionality Tasks model the signal processing operations on the respective IP, whereas Data Transfer Tasks account for the behavior of the local DMA of the corresponding IP. The tasks executed on the LEON processor are henceforth referred to as control tasks. They are not supposed to perform extensive computations or data transfers. Instead, they delegate the work to a task running on a hardware accelerator of the DSP module, hence a task suffixed with "_Func". This is achieved with DIPLODOCUS synchronization mechanisms, events and requests. In the following, whenever it is stated that a control task reads/writes data, it is implicitly assumed that this task relies on the service of a Data Transfer Task.
At the end of this design stage, our UML component diagram (or an equivalent class diagram) has been populated with tasks deduced from the specification.

8.2.2.2 Abstracting communication

After the functional analysis, emphasis is now placed on the the communication behavior of the previously identified tasks. The designer is faced with the question which of the available communication primitives best captures the data dependencies and the impact of data transfers on performance. As a rule of thumb, events transfer control information but no data and the opposite case (conveying data but no control) is handled by non blocking write/non blocking read (nbrnbw) channels. The other channel types are in between these extreme cases. In the model, we opted for a proper separation of control and data flow. We relied solely on nbrnbw channels and operations on these channels are synchronized by means of events. Moreover, abstract data samples need to be accompanied by meta data as described previously. In DIPLODOCUS, this scenario has to be captured by a combination of a channel and an event anyway. Recall that abstract channels do not convey values at all. A detailed discussion of communication links needed in the model would not bring new methodological insights and is therefore omitted.
Hence, after having accomplished this stage, the component diagram has been enriched with communication links between tasks.

8.2.2.3 Behavioral description

The next steps towards an executable model is to detail the behavior of each task by means of a UML activity diagram. The global system behavior defined in the specification must therefore be broken down into suitable local behavior descriptions for each task.

The model includes a test bench task called *PacketGenerator* which compensates the lack of real data samples. This task generates random variables for data length, inter packet arrival time and modulation and forwards them to the *PacketDispatcher*. Moreover, the *PacketGenerator* regularly sends events to the *PacketDispatcher* signaling the arrival of new data samples from the antenna. *PacketDispatcher*, as its name suggests, notifies the availability of data samples to the respective tasks of the control module. That way it captures the relevant behavior of the Preprocessing IP on the real system, which regularly sends interrupts to the control module.

Figure 8.3 gives an overview of the heart of the model, namely the control tasks directing the decoding of a portion of the 802.11p packet (cf. Figure 8.2). All depicted tasks are supposed to run on the control module, hence the Leon processor. As mentioned earlier, these tasks merely configure Functionality Tasks (suffix "_Func") mapped onto the dedicated IP block of the DSP module (like FEP, Channel Decoder, Interleaver) and initiate data transfers.
The *Synchronization* task evaluates the STS field of a packet. Its main purpose is to distinguish noise from incoming data, and to detect the beginning of a new packet. The synchronization to the input stream is achieved by energy detection and correlation of the input samples with a known reference. Requests for DSP operations are sent to the *FEP_Func* task
The *ChannelEstimation* task exclusively considers the LTS field of the packet. Its goal is to correct the amplitude distortion caused by the transmission channel, by comparing the spectrum of a known signal to the one of the received samples. The two aforementioned tasks rely solely on the FEP IP represented by the *FEP_Func* task.
Subsequently, task *SignalFieldDetection* leverages the SIGNAL field of the packet to carry out a phase correction, by applying the already identified correction factor for the amplitude. In a real system, meta-data such as the length of the data payload and the modulation scheme is extracted from the signal field. In our model, we randomly determined the two values in the *PacketGenerator* task, as stated initially. *SignalFieldDetection* accounts for the cost of decoding the meta-data by sending appropriate requests to *FEP_Func*, *ChDec_Func*, *Deint_Func*.
The decoding of the payload ("DATA" symbols in Figure 8.2) is decomposed into three stages, each of which exclusively uses one resource out of the three IPs Front-end Processor, Deinterleaver, Channel Decoder. The stages are represented by a dedicated control task prefixed with "DATA" and may be pipelined for consecutive DATA symbols. As a consequence of their resource usage, *DATA_FEP* only communicates with *FEP_Func*,

DATA_Deint with *Deint_Func* and *DATA_ChDec* with *ChDec_Func*.
DATA_FEP initiates phase and amplitude correction of data symbols and stores results in a global memory to make them accessible to the *DATA_Deint* task. The latter sends requests to the *Deint_Func* task according to the modulation algorithm, issues the processing and again copies results to a global memory. Last but not least, the *DATA_ChDec* task copies data to be decoded to the internal memory of the Channel Decoder IP, initiates the decoding procedure by sending a request to *ChDec_Func*, and reads back the results.

The next section reveals how the architecture of the ExpressMIMO-Card (compare Figure 8.1) translates into a DIPLODOCUS architecture and mapping model, the last step in the methodology.

8.2.2.4 Architecture

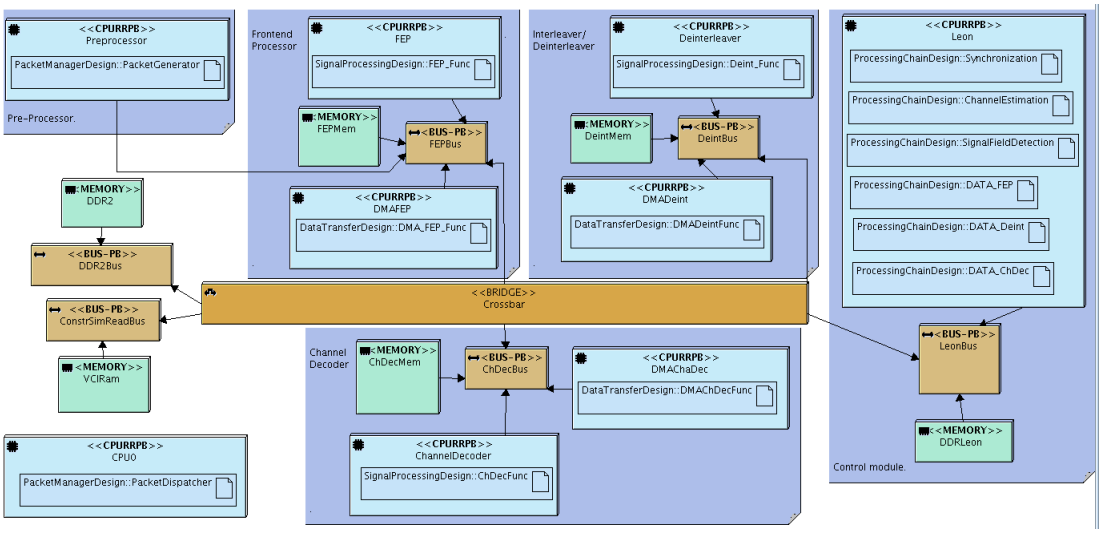


Figure 8.4: DIPLODOCUS Architecture of the 802.11p receiver

In Figure 8.4 it can be seen that the IPs of the DSP module share the same generic structure. They are represented with two CPUs, one accounting for the processing engine itself and another CPU running the DMA functionality, an internal memory and an internal bus interconnecting the aforementioned components. Two global memories, named *DDR2* and *VCIRam* are accessible to all IPs and are used to store intermediate results. The control module consists of the Leon processor onto which the tasks depicted in Figure 8.3 are mapped, as well as a local memory. All IPs communicate by means of a crossbar, which was simply modeled by interconnecting local buses with each other via an intermediate bridge. For the sake of clarity, the mapping of channels onto memories and buses has been omitted.

8.2.2.5 Discussion

The model of the decoder can be considered as a representative of a whole family of signal processing applications. The pattern was basically characterized by three elements: (1) a control task delegating workload to specialized tasks, (2) the latter being chained consecutively with events and requests and (3) meta data that is generated in a dedicated task with non-deterministic operators and subsequently propagated among the processing tasks. We are confident that this pattern can be reused for future case studies in the signal processing field. The model also suggests that even in traditionally data flow dominated domains control flow is getting more and more involved. That makes an even broader range of domains amenable to an analysis with methodologies like DIPLODOCUS.

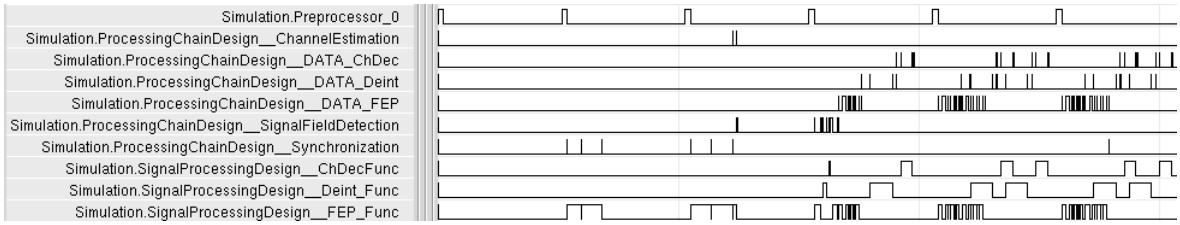
8.3 Experimental results

This section highlights the achievements of this work and explains how they are used in practice. The developed simulator now supports sophisticated architectures including multiple CPUs, (multi channel) buses and bridges, which was not the case for the former one. Simulation outcomes in the form of waveforms are exemplified in section 8.5. Prior to this work, system properties could only be specified informally in the form of Requirement Diagrams. The expressiveness and ease of use of TEPE is illustrated in section 8.3.1.4 with common system properties. Coverage enhanced simulation, another contribution of this work, is successfully applied in section 8.3.2 to partially cover different valuation of non-deterministic operators in the model.

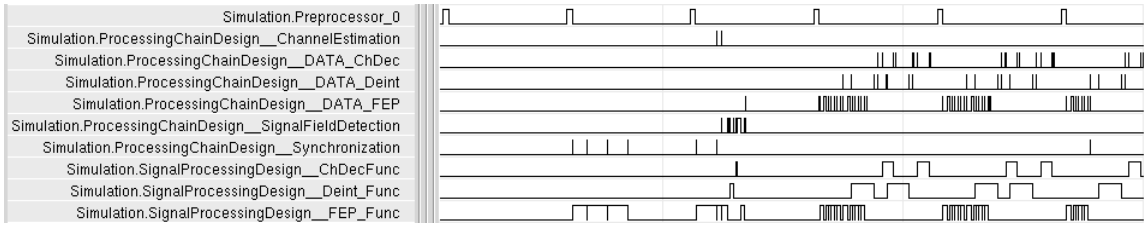
8.3.1 Functional properties

8.3.1.1 Simulation

The waveform depicted in Figure 8.5 illustrates the activity of components when treating two 64 QAM (rate $\frac{3}{4}$) packets comprising 5 data symbols. Note that the time line has been cut off for space reasons as it is not relevant for the following qualitative analysis. The time line normally provides first insights into the timing and duration of actions. A waveform also allows developers more familiar with hardware description languages to spot easily unexpected behaviors and to correct the model accordingly.
The first row shows the arrival of new data samples from the Preprocessor, each edge stands for 160 samples. The subsequent six rows depict activity periods of tasks running on the control module: *ChannelEstimation*, *DATA_ChDec*, *DATA_Deint*, *DATA_FEP*, *SignalFieldDetection*, *Synchronization* tasks. Finally, the last three rows correspond to tasks mapped onto the processing IPs, namely *ChDec_Func*, *Deint_Func* and *FEP_Func*. It is assumed that both packets are preceded by 100 non significant noise samples. As the initial FFT operates on 256 samples, the procedure can start as soon as two times 160 samples are received from the *Preprocessor* task. The first two peeks of the *FEP_Func*



(a) First packet . . .



(b) . . . second packet

Figure 8.5: Simulation result for two 64QAM (rate $\frac{3}{4}$) packets

waveform denote the energy detection and correlation procedure. Thereafter, the small peaks in the waveform of *FEP_Func*, *FEP_Func* and *ChDec_Func* indicate the analysis of the SIGNAL field. The five data symbols are decoded once the corresponding modulation scheme is known. The figure shows five contiguous peaks for the *DATA_ChDec* and *DATA_Deint* tasks, whereas the five activity periods of the *DATA_FEP* are interrupted due to necessary data transfers and other operations. Note that that consecutive data symbols are handled in a pipelined fashion by the three processing tasks. The processing stages may thus overlap to reduce the execution time. After the last data symbol has been processed, the system enters once again the energy detection and correlation mode in order to wait for the arrival of the next packet. Throughout the figure, it is obvious that each activity of a task running on an IP is preceded by an activation period of a control task launching the former.

8.3.1.2 Design Space Exploration

To convince the reader that the simulator is a useful tool for Design Space Exploration, several modifications of the original architecture have been examined. In the following, we consider the decoding procedure of a 64 QAM (rate $\frac{3}{4}$) packet comprising 100 data symbols. At first, the crossbar, represented by a bridge in Figure 8.4, has been replaced with a simple bus. The average contention delay denotes the average time experienced by masters waiting for the arbitration of an interconnect device. The modification of the architecture did not have a notable impact on the average contention delay experienced. This is easily explained with the fact that data transfers are sequential for this communication standard to a large extent. This would surely be different when running several

communication standards in parallel.

Furthermore, we ran the model with various global clock rates (the same for all components) between 1 and 150 MHz. Our objective was to find the smallest frequency that does not violate the assumption that a packet is completed before the arrival of the subsequent one. Therefore, we assumed a minimal inter packet spacing of 10 us. We obtained a frequency of 136 MHz, which may guide the developers in their attempt to fix the global frequency of the system.

To illustrate the significance of the parametrization of hardware components, we compared the load of an ideal processor (disabled penalties) for the control module to a processor with enabled penalties. A task switching time of 10 cycles, a branch prediction failure rate of 10%, and wake up delay from power saving mode of 10 cycles lead to an increase in load of 16% (from 7% to 23%).

One benefit of the methodology is to quickly change the allocation of processing elements to tasks, called mapping. This is simply accomplished in a drag and drop fashion. However, for a lack of calibration data, we were not able to experiment with different mappings. As defined in 2.5, calibration adjusts generic hardware models to the particularities of a real device. If the complexity of an FFT or correlation operation on the Leon processor were known, processing tasks could be mapped onto the latter. In so doing, software and hardware implementations could be traded-off against each other.

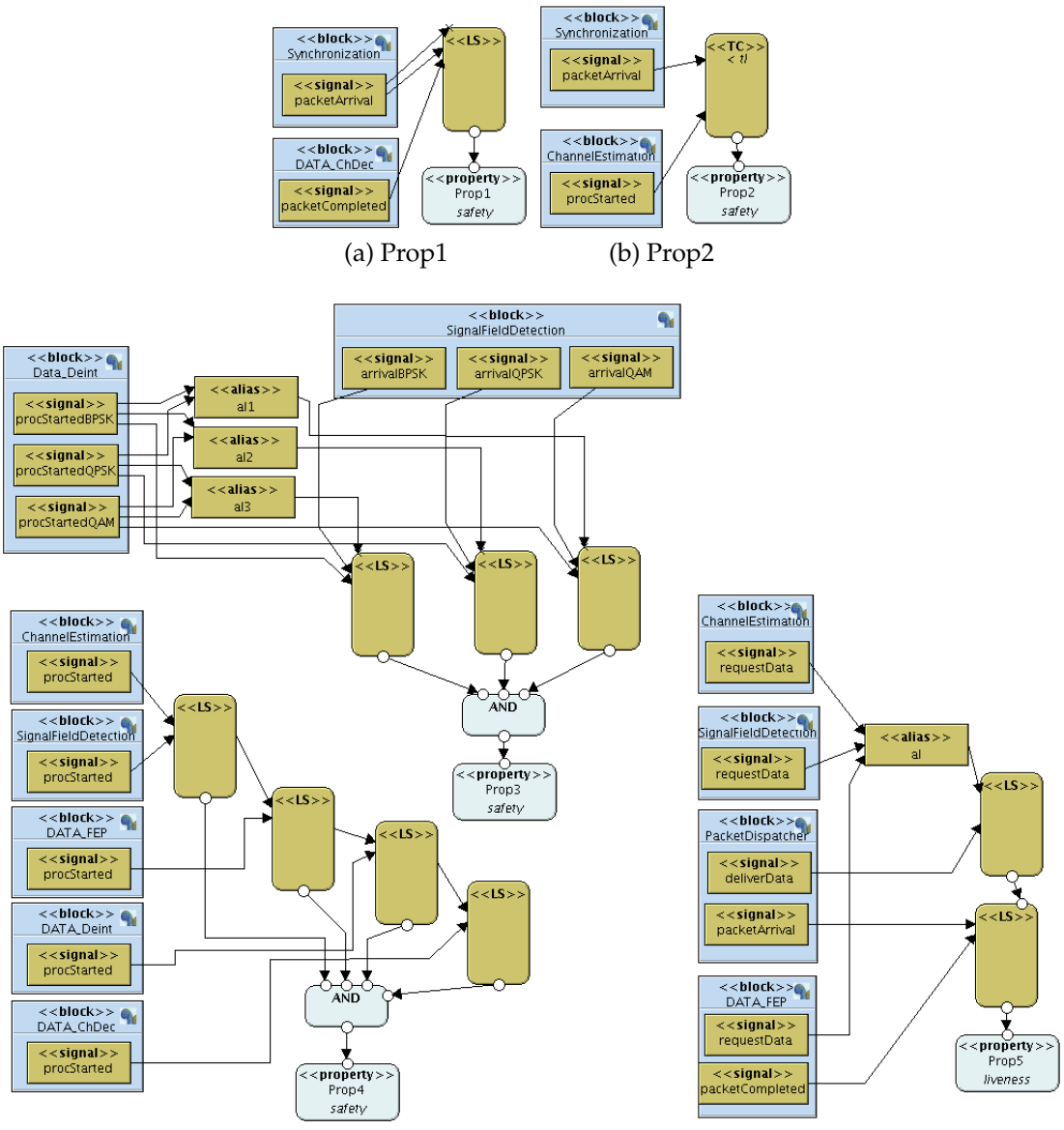
8.3.1.3 Discussion

The former simulator merely supported single processor systems comprising one bus and one memory element. Thanks to the extension of simulation semantics, much more sophisticated architectures may be simulated (see Figure 8.2.2.4) with a better performance. The simulator supports the DIPLODOCUS Design Space Exploration paradigm.
For previous studies, simulation was carried out at $20.25\frac{cycles}{trans} \cdot 0.782\frac{Mtrans}{sec} = 15.83\frac{Mcycles}{sec}$, while performance figures turned out to be very close to results obtained with time annotated functional models (roughly 5-10% deviation).

8.3.1.4 TEPE diagrams

This section discusses relevant properties of the receiver that were defined in close collaboration with a domain expert. Figure 8.6 depicts the TEPE properties to be verified on the model of the 802.11p receiver. We assume a table relating DIPLODOCUS and TEPE entities (cf. section 4.5.2) to be given. Informally, the diagrams express the following conditions to be met:

- **Prop1**: The **deadline** of packet processing is the arrival of a new packet, that means that there is always at most one packet being processed in the system. A Sequence



(a) Prop1 (b) Prop2

(c) Prop3, Prop4 (d) Prop5

Figure 8.6: TEPE properties to be verified

Constraint is configured such that two consecutive occurrences of the *packetArrival* signals sent by the *Synchronization* task without an intermediate *packetCompleted* signal sent by the last task in the processing chain (*DATA_ChDec*) are considered as a failure case.

- **Prop2**: The **latency** of packet processing, defined as the time between detection of the packet by the *Synchronization* task (*packetArrival* signal) and the activation of the *ChannelEstimation* task (*procStarted* signal) is bounded by *tl*. The property is easily expressed with the TEPE time constraint.

- **Prop3**: The modulation type the *SignaFieldDetection* task has identified and the demodulation algorithm chosen by the *DATA_Deint* task should agree. To this end, three Sequence constraints monitor the signals that announce the arrival of packets whose payload is modulated with BPSK, QPSK and QAM respectively. Depending on the arrival signal, only one of the three signals denoting the execution of the *DATA_Deint* task does not indicate a failure and is required. The other two are combined with an Alias Constraint to a compound signal that reveals a failure case. The output of the Alias constraint is thus connected to the negated signal input of a Sequence constraint. The semantics of the Sequence constraint also implies that an *arrival* signal is eventually followed by a *procStarted* signal.

- **Prop4**: If the first processing task (*ChannelEstimation*) is started, then all following tasks (*SignalFieldDetection*, *DATA_FEP*, *DATA_Deint*, *DATA_ChDec*) are executed as well. This example demonstrates the series connection of n Sequence Constraints behaves like a Sequence Constraint with $n + 1$ inputs. This is made possible by the output signal of a Sequence Constraint, which is notified upon a correctly observed sequence of the two input signals. The property output of each Sequence Constraints have to be logically combined with an AND to derive the final result.

- **Prop5**: This constraint is less relevant for the final system but important to the DIPLODOCUS model. As stated previously, the latter comprises tasks for packet generation (*PacketGenerator*) and distribution (*PacketDispatcher*). *PacketGenerator* periodically informs *PacketDispatcher* about the arrival of new data samples. The intention of this property is to guarantee that sufficiently many data samples are produced to process a given number of packets. Otherwise the model could get stuck just in the middle of a packet. A processing task (*ChannelEstimation*, *SignalFieldDetection*, *DATA_FEP*) attempting to read data from the wireless channel should eventually be granted new samples. The lowermost Sequence constraint limits the scope of the upper one to the period bounded by a *packetArrival* signal, sent by the *PacketDispatcher* and the *packetCompleted* signal, sent by the last task in the chain (*DATA_FEP*) that reads data samples. The upper Sequence operator now makes sure that any *requestData* signal, stemming from *ChannelEstimation*, *SignalFieldDetection* or *DATA_FEP* is concluded by a *deliverData* signal of the *PacketDispatcher*.

8.3.1.5 Discussion

The provided examples cover some general types of constraints that are frequently encountered when verifying high level models of SoC. The first two properties (Prop1 and Prop2) refer to a deadline or latency to be respected. This is important to systems subject to real time system constraints. However, when modeling the latter with DIPLODOCUS, one has to bear in mind that abstraction entails an inaccuracy of results. Prop3 illustrates the common case in which the occurrence of a signal must eventually be concluded by the concurrence of another one. This is the liveness part of the property. In addition to that, some other signals are considered as indicators of a failure case, which makes up the safety part of the property. Prop4 falls into the category of properties where signals must be received in a specific order, which may be relevant for the scheduling of data dependent actions. Finally, Prop5 realizes the nesting of intervals. This is especially interesting if systems may function in different modes, that determine the scope or relevance of properties.

8.3.2 Non-functional properties and coverage enhanced simulation

This section puts emphasis on the benefits of coverage enhanced simulation. Thanks to the aforementioned features, the inherent indeterminism of the application model can be exploited at the push of a button. Simulation results are presented in a form which is easily accessible to developers of the signal processing community. This provides evidence for the versatility of the DIPLODOCUS approach. Outcomes of simulation are not limited to summary tables with performance figures and reachability graphs (cf. section 7.4), being definitely ill-suited for visualization in this case.
As stated initially, the ExpressMIMO-Card will finally accommodate several telecommunication standards which may also operate in parallel. With the hardware architecture already dimensioned, the platform is currently simulated with different standards alone. Performance measures can then be leveraged to spot a suitable scheduling strategy in case several standard share the hardware resources. This study aims to give insights into the resource consumption of the 802.11p standard.
Figure 8.7 has been established with results directly obtained from the simulator, that were plotted in Matlab. In the figure, the utilization of components is plotted against the number of data samples per packet for each modulation standard in the same diagram. Three distinct diagrams visualize the utilization of FEP, Deinterleaver and Channel Decoder respectively. The common ground of all curves is that departing from an initial utilization where only one data symbol is sent, they eventually reach a steady state. The initial utilization is largely determined by the computational power required to detect the start of a new packet. That is why it does not differ much between different standards for one processing IP. When moving towards large packet sizes, the impact of packet detection decays while data processing starts to predominate resource utilization.
For Channel Decoder and Deinterleaver, processing gets more demanding from BPSK (rate $\frac{1}{2}$) to 64 QAM (rate $\frac{3}{4}$), with modulation schemes being ordered in the same way

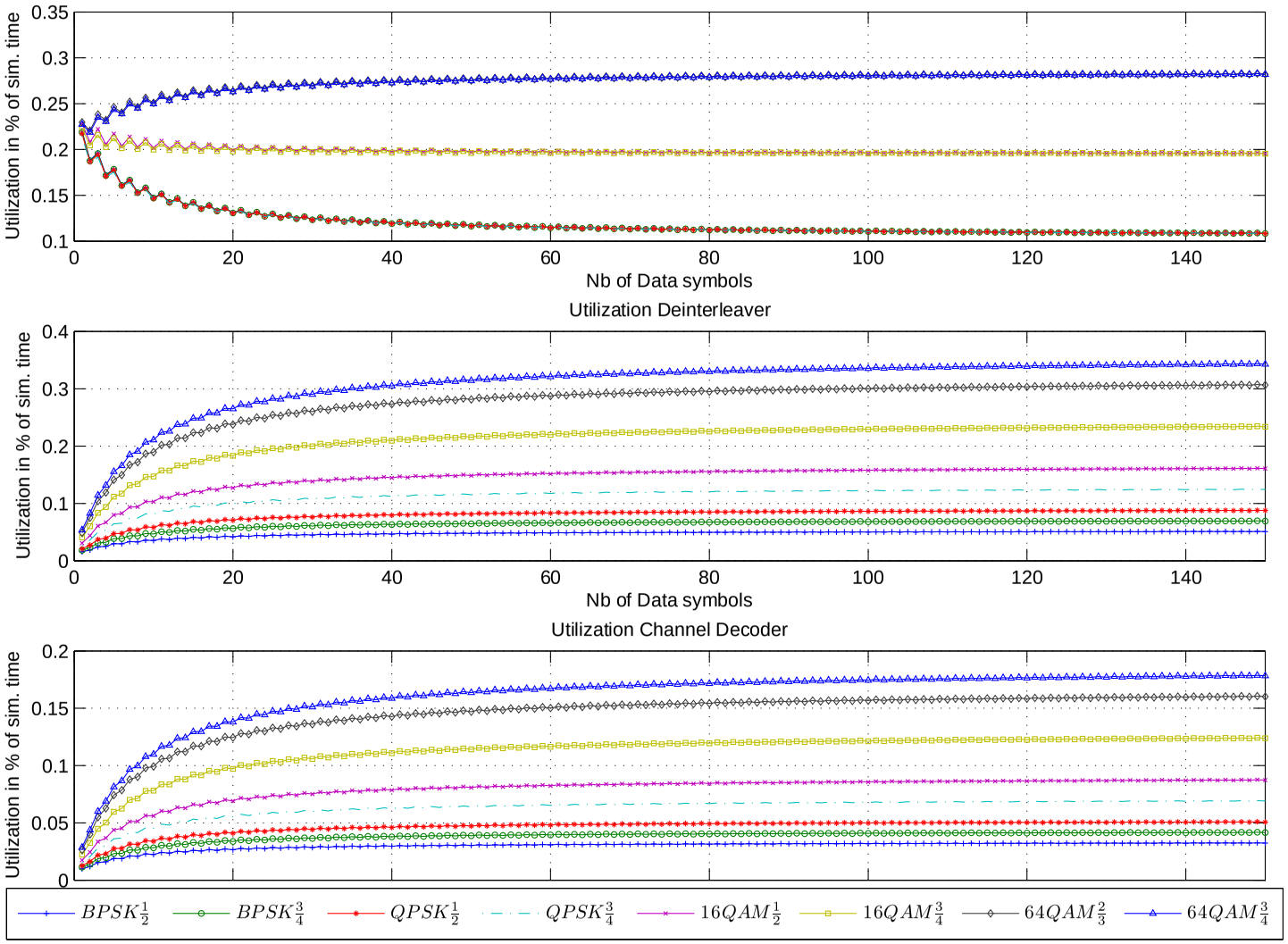


Figure 8.7: Utilization of Hardware Components

as in the legend of Figure 8.7. In both cases, large packets are more demanding than smaller ones. This is different for the FEP IP. As it is heavily involved in packet detection, decoding BPSK and QPSK (lower curve) symbols is far less costly and 16 QAM (graph in the middle) a bit less costly than the initial procedure, the corresponding graphs decay. However, the graph for 64 QAM exhibits the same qualitative trend as the graphs for the other IPs. The slight ripple of all curves can be explained with the periodic arrival of data samples. Resource utilization locally decreases when the receiver has to wait for one interrupt more in order to gather the data. With t_w denoting the waiting time for new data and t_p the overall additional processing time, simulation time increases by $t_w + t_p$ whereas the utilization time of an IP increases by at most t_p, hence less than $t_w + t_p$. In the same way, we could have analyzed the idle time of components so as to figure out laxity permitting to execute other standards.

8.3.2.1 Discussion

The previous study was made possible by the following features that are an achievement of this thesis:

- extending the coverage of verification while preventing a conversion to formal models. As stated in section 6.1, representing sophisticated mapping models is cumbersome in formal language. We experienced that models are rejected by UPPAAL and CADP model checkers and that the acceptable processing time or memory is exceeded.
- selecting the non-determinism of the application model to be exploited without having to recompile the model. Prior to this work, UML models had to be modified and formal models to be regenerated in in order to vary the coverage.
- using a step width for valuations of random commands to be explored to prevent state explosion. Especially for random commands spanning a huge range of values, the new method makes it easy to cover only a subset of them.
- offering a convenient automatization of the design flow if non-determisism is spread over several tasks. When compared to a manual exploration of different executions, the new method prevents one from searching for an indeterministic command in the model, modifying the model to chose one valuation, recompiling and relaunching the simulation.

Coverage enhanced simulation was carried out at $14.54\frac{cycles}{trans} \cdot 0.272\frac{Mtrans}{sec} = 3.95\frac{Mcycles}{sec}$, which includes the performance impact of state hashing.

8.4 Conclusions

In the case study, we have applied the contributions introduced in previous chapters such as the formulation of properties in TEPE, the conventional simulation of DIPLODOCUS models and the coverage enhanced simulation to a realistic problem. To this end, we relied on the tool chain whose simulation features were developed in the scope of this thesis. The section drew upon the example of an 802.11p receiver which is currently deployed on Eurecom's ExpressMIMO-Card. The latter implements the Software Defined Radio paradigm. In the introduction, we made the reader aware of the intricacy of the task of hosting several wireless standards on the same platform, sharing the same resources. Therefore, the need arose to build a high level model to study the resource utilization of the 802.11p receiver. It was demonstrated that coverage enhanced simulation alleviated this task considerably while preventing state explosion. With the valuable assistance of a domain expert, it took us merely two weeks to familiarize with the problem domain and to conceive the model. This time, the hardware platform had already been specified, making experiments with different architectures obsolete. Instead, it was demonstrated that outcomes may guide the designer in his effort to spot a suitable scheduling involving more than one communication standard. Moreover, TEPE was very capable when it came to expressing functional properties being intuitively understandable by less experienced designers.

Chapter 9

Conclusions

In the introduction, we identified the need for modeling environments that capture embedded systems and Systems-on chip at a high level of abstractions. That way, the designer may obtain first insights into the behavior of the system under design even if low level models are not available or too costly to build. Similarly, it could be of interest to abstract detailed models in order to speed up simulation or to extend its scope (for instance from module to system wide). Abstractions are essential for Design Space Exploration, where a large number of implementation alternatives are to be investigated. In chapter 3, we examined some representative samples of the huge body of related work. The categorization of work revealed the fundamental difference of static and dynamic models, the latter necessitating an implicit or explicit representation of system states. DIPLODOCUS diagrams model behavior in a state based fashion and thus (in general) necessitate a traversal of the state space to be verified. This thesis is devoted to the execution and verification of high level models exhibiting a similar semantics to DIPLODOCUS. Section 9.1 reviews the contributions of this thesis. In section 9.2 we revisit shortcomings of related work identified in section 3.7 and our initial claims. We summarize the benefits of DIPLODOCUS in general and this work in particular and relate them to examples provided throughout this thesis. Section 9.3 surveys further research that could complement the presented solutions.

9.1 Resume of Contributions

TEPE
We have proposed the verification language TEPE, which aims to bring together design and verification at the same level of abstraction. Emphasis is placed on relations of properties and signals in the TEPE terminology, or in other terms states and events. The language may be conveniently used in conjunction with UML/SysML and other models of computation (MoC) originating from labeled transition systems. TEPE merely presupposes the existence of primitives similar to signals and events. It allows both design and verification to be seamlessly accomplished in the same environment. Our language

enriches conventional design methodologies with a different formalism preventing that errors in reasoning propagate from design to verification stage. Traditionally, verification of high level models is still conducted in temporal logic, an error prone undertaking even for experienced designers. TEPE attempts to make formal verification more intuitive while being formally defined in terms of fluent logic (FLTL). Constraints may emulate non-observable states and signals of a system. In this thesis, we have successfully applied TEPE to two UML profiles, AVATAR and DIPLODOCUS. Moreover, we demonstrated how informally defined requirements translate into TEPE. Thereby, the model of a microwave oven and an 802.11p decoder served as an example. We further elaborated on a possible implementation of a TEPE verifier which could be operated on the fly during simulation or off-line on traces.

High Level Simulator
A simulation framework was conceived that is especially suited for high level models featuring data abstraction and symbolic instructions, like DIPLODOCUS. It was argued that SystemC concurrency and synchronization primitives exhibit a sophisticated semantics which is applicable to a broad variety of applications at different levels of abstraction. DIPLODOCUS semantics is however far less involved, restricting for instance communication to multi-point to point and preventing zero delay feed-back loops. Thus, we realized a gain with a simplified simulation semantics, thus renouncing the genericity of SystemC and tailoring discrete event simulation to the needs of high level models. To this end, the simulation strategy avoids an explicit event queue and limits the number of active components to the number of CPUs. Only active components may propose transactions spanning possibly many clock cycles. Transactions are subject to a lightweight postponing and truncating policy, thus circumventing time consuming operations on an event queue.
The last part of chapter 5 elaborated on implementation issues and compared the performance of the new simulator to the former one existing prior to this work. It turned out that performance in terms of cycles per second is composed of a model and implementation dependent part. An achievement of this work is that a coarse granularity of the application positively impacts the performance. Even in the worst case when the average transaction length amounts to one, we experienced a gain of factor 10 with respect to the SystemC based simulator.

Coverage enhanced simulation
The simulation methodology was extended to cover several control flow branches of the application. In environments relying exclusively on simulation, spotting suitable test vectors that provide an acceptable coverage of the model is difficult. DIPLODOCUS has the nice property of making data dependent decisions explicit with non-deterministic operators. Prior to this work, formal verification already leveraged these operators to explore all possible executions. However, the translation of mapping models resulted in complex syntactical structures impeding the verification with UPPAAL and CADP model checkers. Moreover, once the UML model is transformed into a formal language,

coverage is static and may not be varied any more. Coverage enhanced simulation addresses (CES) the mentioned shortcomings. Coverage criteria, being themselves of scope of this work, can be taken into account on the fly at simulation time. In the case study, a series of utilization curves was easily established thanks to a partially covered application model.

Tooling
Finally, a prototype implementing the aforementioned contributions was realized. Section 7.4 covered the design flow with TTool, the automated model transformation to executable C++ code, interactive simulation for debugging purposes and the communication of the graphical interface and the simulation engine. TTool was enhanced to support fast simulation, CES and basic verification features for TEPE diagrams.

9.2 And finally. . . - initial claims revisited

To complete the review of DIPLODOCUS in general and contributions of this work in particular, we get back to the starting point of this report. The incentive for this work directly resulted from shortcomings of existing approaches (cf. section 3.7). Based on the insight gained in previous chapters, we can now give more concrete examples of how these shortcoming are remedied:

- *Purely analytical (data flow) models tend to be overly pessimistic or optimistic (WCET BCET) and do not detail control flow within tasks at all. . .*
  Even if our approach relies on data and abstraction and symbolic operations, control flow is represented to some extent, as it was demonstrated in the case study. Behavioral task descriptions allow us to generate waveforms or Gantt diagrams that visualize system behavior in an intuitive way. In our case study for instance, results were easily understandable by a domain expert not familiar with DIPLODOCUS. Nevertheless, she was able to suggest modifications of the model.

- *In some frameworks, models need to be refined (using code) before being simulated which entails an increased modeling effort and badly affects simulation performance. . .*
  An estimation of the accuracy of the DIPLODOCUS method is out of scope of this thesis. It should however be noted that performance figures obtained in the case study came close to the analytical calculations of domain experts. This was achieved with a calibration based on estimations originating from low level models. Design Space Exploration does not require a refinement of DIPLODOCUS models. While capturing the main performance aspects of the application, the model nevertheless simulates efficiently, as shown in the case study.

- *Almost all simulation frameworks rely on SystemC and thus tend to make use of detailed architecture models which are very costly in terms of simulation time. . .*
  In chapter 5, we provided evidence for the fact that the DIPLODOCUS semantics

can efficiently be expressed in C++ without relying on SystemC. The simulation strategy introduced in this work has proven to be faster than the former SystemC based simulator.

- *Many employed MoCs (KPN, data flow networks,..) do not make data dependence or non-determinism explicit at application level. . .*
  The case study (section 8.3.2) highlighted the strengths of coverage enhanced simulation. Due to data abstraction, data dependent decision are modeled with indeterministic operators in DIPLODOCUS. CES permits to automatically explore all valuations of random variables to study their impact on system performance. In the case study, we plotted the utilization of different IP blocks as a function of random variables to exemplify the approach.

- *To our knowledge, state of the art UML model simulators target a solely functional verification of the system. . .*
  Embedded systems by definition involve computation that is subject to physical constraints. The case study revealed two kinds of constraints: on the one hand, the presented decoder had to react to its environment, namely packets conveyed by the wireless channel. On the other hand, the predetermined architecture imposes constraints on possible scheduling policies. The assumption of abundant hardware resources cannot be maintained in the field of embedded systems. The DIPLODOCUS methodology acknowledges this fact by explicitly considering these constraints in the model, independently from the application.

- *The additional modeling effort inherent to some other approaches is mitigated in our case by the usage of UML models and automated model transformations. . .*
  All in all, it took us around two weeks to get familiar with the problem domain of the case study and to build the model. Less than one week was allotted to the construction of the model alone. A domain expert with a basic knowledge of DIPLODOCUS would have accomplished that task even faster. We think that the results definitely justify the effort, especially compared to the effort which would have been spent on low level modeling. For the time being, DIPLODOCUS models cannot be refined for code generation. Ongoing research work will remedy this problem in the near future.

- *Many environments do not fathom the trade-off between exhaustive formal and simulation techniques*
  In the case study, we opted for a partial coverage of the model by only considering the processing of one single packet. The exploration of series of packets would not have yielded new insights into the utilization of IP blocks, as the procedure for packets of the same type is always the same. Moreover, based on our results, the utilization of components for a series of packets could be determined analytically.

- *Even though some frameworks are settled at a high level of abstraction, verification relies on obscure logical formulas. . .*

In both presented case studies (microwave oven and 802.11p decoder), the formulation of properties in TEPE turned out to be intuitive and has been feasible for a designer with minor experience in temporal logic. TEPE directly refers to model elements and captures properties at an equivalent abstraction level to DIPLODOCUS.

9.3 Limitations and Future Work

Attention is now turned towards future work complementing this thesis and aspects the author would have liked to deals with, but could not for a lack of time.

9.3.1 Methodological Aspects

Targeting code generation and model refinement
Even if the construction of DIPLODOCUS models is not time consuming, it would be desirable to leverage this formal specification for code generation purposes. Further research has to reveal under which conditions functional properties proved in DIPLODOCUS are conserved at lower abstraction levels. That way, the tedious tasks of defining memory maps and the control part of a system could be considerably alleviated. However, it is unlikely that complete systems may be entirely generated at the push of a button in the near future. For this reason, guidelines have to be pointed out stipulating how the generated code may be manually refined without disrupting the relationship with the high level model.

Drawing analogies between performance and power consumption
The idea of assigning costs to symbolic operations could apply equally well to the realm of power consumption. The current model of a power manager only penalizes tasks that become runnable after a certain idle time of the CPU. The implications of voltage and frequency scaling have not been taken into account so far. To start with, one could just assign an energy quantum to both operations and idle time of hardware components. To calculate the energy consumption of an entire system, energy quanta are summed up per operation and per involved hardware component. In a second stage, the approach could be extended to a more complex power manager modeled with a DIPLODOCUS activity diagram. If timing aspects are irrelevant, one could perhaps come up with a simplified simulation procedure especially suited for power consumption. This procedure could save time by disregarding the partial order of some operations, as long as power consumption is not impacted.

Reinforcing modularization
In the future, DIPLODOCUS component diagrams should permit to reinstantiate components once defined. This would especially be useful in applications that involve recurring structural patterns. For the time being, common behavior within tasks or

across several tasks has to be externalized in another task. This requires additional synchronization primitives, that could provoke inconsistencies with low level models. If the latter are endowed with macro like constructs, the synchronization primitives could vanish during the refinement process. DIPLODOCUS should consequently offer parametrizable macros that are just expanded before model transformation. This would even leave DIPLODOCUS semantics unchanged.

Introducing abstract, composable building blocks to model communication architectures
The multi-channel buses implemented in the simulator can be used to compose much more sophisticated communication architectures. In the case study for instance, a crossbar has been adequately represented with buses, links and bridges. Architecture diagrams should model communication media in a much more composable and abstract way, merely distinguishing the capacity of a link in terms of simultaneous transactions. An infinite capacity could be assigned to a crossbar, where concurrency is only limited by the number of communication targets reachable by initiators.

9.3.2 TEPE semantics

Thoroughly analyzing expressible properties
Let us assume that simulation start and termination have their representation in terms of dedicated signals, s_{start} and s_{end}. In that case, we can come up with a TEPE constraint expressing that *something bad (s_n) never happens*. This could be realized with a Sequence Constraint with s_{start} and s_{end} as input signals and s_n as negated signal. We may further formulate that *something good (s_2) eventually happens*, for instance with a Sequence Constraint applied to s_{start} and s_2. This thought experiment suggests that TEPE supports both liveness and safety properties. However, the claim should be supported in a mathematically rigorous manner. The objective is to conclude that TEPE may express any property that temporal logic can, based on the decomposition theorem for properties.

Establishing the relationship between TEPE Sequence Constraints and Regular Languages
As we succeeded implementing a TEPE verifier in terms of finite state machines, there is a strong evidence that Sequence Constraints define a regular language. To reinforce the formal underpinning of TEPE this relationship should be further investigated. Sequence and Logical operators could be generalized into extended regular expressions, that way increasing the expressive power of the language. As a matter of fact, we will not be able to formulate properties such as *there are as many occurrences of s_1 as of s_2*. Properties of this type lie beyond the expressiveness of regular languages. Their verification would in general require an infinite amount of memory. Moreover, the expressiveness of TEPE with respect to temporal logic and verification languages such as LTL, CTL, CTL* and PSL shuold be investigated in a mathematically rigorous manner.

Distinguishing simultaneity and periodicity
So far, TEPE Time Constraints cannot distinguish between simultaneity and periodicity. If the same signal is connected to the two input ports, the property only holds if $T_{min} = 0$. For this reason, periodicity of signals cannot be checked for jitter, etc. It should be investigated how to minimize the extension of the TEPE semantics when introducing an additional operator.

Carrying out further case studies with TEPE
The semantics of TEPE should be validated with further case studies and adapted if needed. Only an extensive practical use can expose deficiencies and properties which are difficult to express.

9.3.3 Coverage enhanced simulation

Elaboration of coverage criteria
Sophisticated coverage criteria could be built upon CES proposed in this thesis. Heuristics based on TEPE diagrams could be realized in addition to traditional function, statement, decision and condition driven coverage criteria . For instance, if a property fails upon occurrence of a particular signal, branches triggering the signal could be privileged to disprove the property as soon as possible. All intermediate states of the verifier automata are available and may be consulted for this purpose. In this context, the power of genetic algorithms [39] could be adopted for a target oriented pruning of the state space. A formal approach involving slicing of DIPLODOCUS applications and/or symbolic simulation with respect to a TEPE property could potentially yield a proof of that property while avoiding an exhaustive coverage of the model. For instance, if a random decision does not have an impact on any TEPE property, it is not necessary to explore all valuations of the corresponding random variable.

Minimization of traces
Up to now, branches can only be merged if similar system states are encountered. For example, if loop boundaries of two executions differ, no merge could take place even if loop counters have the same value. Live Variable Analysis would reveal that the loop boundary is significant and to be taken into account for state hashing. This problem could be remedied by post processing the reachability graph (minimization).

Comparison system states
The used state hashing policy an can be subject to collisions in the hash table. In that case, execution branches would be erroneously merged. To resolve collision in the hash table, a hierarchical technique with recursive indexing [52] should be employed. The method is based on the observation that each task can reach only a relatively small number of local states. State explosion stems primarily from the huge amount of possible

combinations of local states. Obviously, if a local state has not been encountered yet, the same applies for the global state. Moreover, state compression is a must to keep memory consumption within reasonable limits.

Convergence of static analysis
It should be established whether the composition of algorithms used for static analysis of DIPLODOCUS tasks always converges towards a steady state. Even if this has already been accomplished for Live Variable Analysis and Reaching definition analysis (cf. [7]), it is not obvious that the composition of the latter and their execution in a loop conserve the convergence property.

9.3.4 Practical Aspects and Performance

Validation of memory models
For the time being, a fixed data rate accounts for the impact of memories on system performance. This model should be carefully validated with respect to volatile and non-volatile memory types prevalent in embedded system design.

Deadlock avoidance for communication architectures
Deadlocks are not resolved in case different CPUs mutually try to reserve buses already reserved by the other one. Perhaps, an adaptation of the priority ceiling protocol could produce relief.

Enhancement of simulation performance by avoiding too many scheduling rounds for small transactions
For an average simulation run, scheduling accounts for about 10% of the simulation time. This comes from the fact that every transaction is scheduled, even if its duration is short. The overhead could be reduced if a latency t_l is accepted between the activation of a task and its consideration by a scheduler. t_l would then denote the maximum cumulative duration of transactions to be executed without intervention of a scheduler.

Reduction of the overhead due to Action/Choice commands
The simulator was structured with an inheritance hierarchy so as to simplify maintenance and extension. Lightweight operations like Choice and Action are subject to an overhead as the respective function pointers are encapsulated into a class as well. A performance gain could perhaps be realized by moving away from a strictly object oriented architecture.

Automatic exploration of several architectures
It would be desirable to explore a set of architectures automatically, without having to modify the graphical model and to relaunch the simulation. Thanks to the intermediate textual representation of architectures, the latter could be specified before hand

or automatically be generated. A script could then compile the simulator for different architectures, launch it and record the results.

Purely functional simulation without mapping
It could be envisaged to perform simulation of an application model alone. The resulting additional indeterminism could be handled with (1) the proposed CES techniques, (2) a random number generator as in conventional simulation or (3) simply by the user who navigates manually through the model. Together with a state compression and comparison procedure, it could even be envisaged to discontinue the support for formal languages.

9.4 Publications

- Daniel Knorreck, Ludovic Apvrille, and Renaud Pacalet. *Fast simulation techniques for design space exploration*. In Objects, Components, Models and Patterns, Lecture Notes in Business Information Processing, pages 308-327. Springer Berlin Heidelberg, 2009.
- Daniel Knorreck, Ludovic Apvrille, and Renaud Pacalet. *An interactive system level simulation environment for Systems on Chip*. In ERTSS - Embedded Real Time Software and Systems, May 2010.
- Daniel Knorreck, Ludovic Apvrille, and Renaud Pacalet. *Formal system-level design space exploration*. In Proceedings of the 10th IEEE Conference on Distributed Systems and New Technologies (NOTERE 2010), Tozeur, Tunisia, May 2010.
- Daniel Knorreck, Ludovic Apvrille, and Pierre De Saqui-Sannes. *TEPE: A SysML language for timed-constrained property modeling and formal verification*. In Proceedings of the UML&Formal Methods Workshop (UML& FM), Shanghai, China, November 2010.
- Gabriel Pedroza, Ludovic Apvrille, and Daniel Knorreck, AVATAR: *A SymML environment for the formal verification of safety and security properties*. The 11th IEEE Conference on Distributed Systems and New Technologies (NOTERE 2011), Paris, France, May 2011.
- Daniel Knorreck, Ludovic Apvrille, and Renaud Pacalet. *Formal system-level design space exploration*. to appear in: Concurrency and Computation Journal, 2011.
- Currently under review at the Software and Systems Modeling Journal: Daniel Knorreck, Ludovic Apvrille and Pierre de Saqui-Sannes. *Formal and Graphical Modeling and Verification of Temporal Properties in SysML/TEPE*

Chapter 10

French Summary

10.1 Introduction

Les **systèmes embarqués** sont des équipements électroniques dont les unités de calcul sont totalement intégrées à l'appareil qu'ils commandent [109]. Contrairement aux ordinateurs conventionnels à usage général, l'éventail de tâches réalisées par un système embarqué est clairement prédéfini. Aujourd'hui, les systèmes embarqués complexes peuvent être intégrés sur une seule puce et sont alors appelés *systèmes sur puce* (SoC). Les SoC comprennent à la fois un ensemble de composants électroniques communicant et des parties logicielles complexes. Les SoC sont hétérogènes par nature, et comprennent ainsi des composants dédiés au traitement des signaux numériques, analogiques et mixtes. Ces composants sont interconnectés afin de former des systèmes complexes allant des applications mobiles jusqu'aux systèmes de contrôle pour des rames ferroviaires, en passant par des set-top boxes ou des blocs de contrôle électronique véhiculaires. De plus, les densités d'intégration ne cessent de progresser grâce aux progrès dans le domaine de la physique des semiconducteurs, permettant ainsi l'implantation de plus de fonctions logiques dans la même surface de silicium.

Les ressources de traitement et stockage disponibles permettent d'exécuter des applications d'une complexité croissante [35; 49]. D'une part, les utilisateurs exigent des produits présentant des fonctionnalités avancées qui sont fiables, faciles à utiliser et peu couteuses. D'autre part, les développeurs ont du mal à suivre cette évolution en raison des insuffisances des outils et méthodologies. En plus de cette problématique de complexité, le temps de mise sur le marché est devenu un problème majeur. Enfin, les développeurs sont confrontés à des difficultés en raison d'une croissance de complexité importante. En effet, il devient de plus en plus improbable qu'une conception optimale représente une solution intuitive, par conséquent l'expérience du concepteur ne le guide par forcement par rapport aux exigences fonctionnelles et non fonctionnelles requises, notamment en termes de performance, d'espace utilisé, de consommation d'énergie ou de fiabilité. Tout un ensemble de travaux, dont cette thèse, tendent à répondre à la

question de comment la complexité croissante peut être maitrisée.

En considérant une fonctionnalité particulière et les exigences associées, l'**espace de conception englobe toutes les alternatives de conception qui sont fonctionnellement équivalentes** [40]. Étant presque infiniment grand au début du flot de conception, l'espace de conception doit être progressivement réduit en affinant le modèle du système. Moins la spécification est précise, plus le modèle du système présente d'indéterminisme, et plus l'espace de conception est large. Idéalement, les modèles sont progressivement raffinés au cours du flot de conception et résultent en une implémentation qui optimise une fonction pondérée des exigences non-fonctionnelles.

Bien que la procédure semble simple en théorie, **réduire l'espace de conception se révèle difficile à réaliser en pratique.** Des développeurs expérimentés ont tendance à exploiter des plate-forme ou produits qui se sont révélées bien adaptées pour les produits précédents. Ainsi, des incréments souvent mineurs sont apportés à aux anciennes architectures pour en dériver de nouvelles. Si réutilisation de conception se révèle un moyen puissant pour réduire les coûts de développement de produits similaires, réduisant ainsi l'espace de conception, elle n'est pas absolument pas efficace quand il s'agit de trouver une solution proche de l'optimale pour des produits innovants.
Enfin, la conception orientée plate-forme redirige le problème de l'exploration de l'espace de conception du vendeur du produit final vers le fournisseur de la plateforme. Ainsi, la nature du problème reste inchangée et les outils qui permettent d'évaluer des alternatives d'implémentation de la même fonctionnalité sont essentiels.

L'analyse de systèmes à faibles niveaux d'abstraction assure un degré élevé de précision, cependant elle devient exigeante et lente. Les techniques de simulation standard appliquées au niveau transactionnel ou au niveau RTL ne sont pas appropriés pour l'exploration des espaces de conception à un niveau système. Deux raisons à cela:

- Seul un nombre limité d'alternatives d'implémentation peut être examiné en raison de l'effort de modélisation et du temps d'exécution de la simulation.
- Le manque de spécifications à des stades de conception précoces peut rendre difficile - si ce n'est impossible - la construction de modèles détaillés, même si l'effort était finalement acceptable.

En conclusion, l'utilisation d'**abstractions nous parait inévitable** [35] pour réaliser l'exploration à un niveau système et devrait bien entendu faire partie d'une méthodologie de modélisation bien définie. Dans une approche système, l'application et l'architecture doivent être traitées de façon orthogonale. En effet, pour des raisons de réutilisabilité, un modèle d'application ne devrait pas avoir besoin d'être réécrit lors de son expérimentation sur différentes plateformes. Cette stratégie d'exploration est connue comme l'approche en Y [77], et est largement utilisée dans le domaine de l'exploration de l'espace de conception au niveau système.

10.2 Problématique

Le travail présenté dans le cadre de cette thèse recommande des **techniques d'exploration pour faciliter le début du flux de conception d'un système embarqué complexe**. Dans ce contexte, la section précédente a déjà mis en évidence la nécessité d'abstractions. Nous allons maintenant examiner en détail les deux principaux types d'abstraction, qu'ils visent l'aspect fonctionnel ou l'aspect temporel d'un système.

Deux vues orthogonales du système sont couramment répandues. D'une part, le concepteur s'appuie sur des modèles purement fonctionnels et asynchrones afin d'étudier les algorithmes complexes applicatifs. Par exemple, dans le cadre d'un décodeur de modulation d'amplitude en quadrature (QAM), le résultat attendu pourrait être de savoir si le décodeur est capable de reconstruire la séquence originale d'échantillons de signaux arbitraires. Dans cet exemple, la synchronisation des échantillons est considérée comme acquise puisque l'approvisionnement du décodeur avec des données n'est pas pris en compte. L'exactitude fonctionnelle est la seule préoccupation à ce stade. Dans le domaine du traitement du signal et de l'ingénierie de contrôle, des outils mathématiques tels que *Matlab* ont prouvé leur efficacité pour ce genre de prototypage rapide.

D'autre part, il est très important d'avoir une vue globale sur l'interaction des composants du système. Ainsi, l'attention se focalise sur le timing et les performances, ce qui permet de largement abstraire la fonctionnalité. En conséquence, la vue sur l'ensemble du système aide le développeur à réaliser l'architecture du système, c'est à dire à dimensionner les composants et l'infrastructure de communication de sorte que les contraintes non-fonctionnelles soient respectées. Pour revenir à l'exemple du décodeur QAM, l'objectif serait de déterminer quelles implémentations garantissent que le décodeur n'est jamais à court d'échantillons d'entrée et que les échantillons de sortie arrivent à temps à leur destination. Dans ce cas, les calculs internes peuvent être abstraits et représentés par des instructions symboliques: en effet, seul le comportement observé au niveau des entrées-sorties importe dans cette analyse. Ceci nous amène directement à la notion de modèles de performance non-fonctionnels, constituant le sujet principal de ce travail.

Les contributions de cette thèse ont été faites dans le contexte de l'environnement DIPLODOCUS, qui englobe un profil UML, une méthodologie et un outillage. Il est particulièrement adapté pour raisonner sur des modèles de performance abstraits (en termes de fonctionnalités) et focalisés sur le flux de contrôle des SoC actuels.
Cet environnement DIPLODOCUS complète avantageusement les premiers modèles présentés initialement qui sont fonctionnels et untimed. Même dans des domaines qui s'appuient traditionnellement plutôt sur des modèles untimed, l'interaction entre les routines de traitement (du signal) est de plus en plus sophistiquée et doit ainsi être orchestrée par des algorithmes de contrôle. Les environnements académiques et industriels autres que DIPLODOCUS et ayant des objectifs similaires de l'état de l'art

sont souvent limités par les aspects suivants:

- L'application (les fonctionnalités) et l'architecture (la plateforme) ne sont pas traitées de manière orthogonale [60] afin d'accélérer le partitionnement HW/SW.
- L'accent est exclusivement mis sur la seule simulation ou les seules méthodes formelles.
- Un compromis entre ces deux cas extrêmes de vérification n'est pas proposé.
- Les abstractions ne sont pas entièrement mises à profit pour effectuer une simulation rapide.
- L'approche est limitée à des mesures de performance ; le flux de contrôle ne peut pas être modélisé/vérifié.
- Dans la méthodologie, la façon de construire des modèles n'impose pas d'abstractions (par exemple : UML).
- Les détails des algorithmes sous-jacents doivent être fournis sous forme de code source afin d'exécuter le modèle.
- Après la transformation du modèle en un équivalent exécutable, les informations utiles pour le débogage ne sont pas propagées au modèle original.
- Le niveau d'abstraction du langage utilisé pour exprimer des propriétés fonctionnelles ne correspond pas au niveau d'abstraction du modèle du système (dans l'exemple d'un modèle de système en UML, le langage de vérification usité est souvent CTL).

Certains des inconvénients susmentionnés ont été corrigés grâce à DIPLODOCUS lors de travaux antérieurs à cette thèse : ils font l'objet de la prochaine section.

10.3 Objectifs et Contributions

Cette thèse est consacrée à l'amélioration d'un environnement existant d'exploration de l'espace de conception qui est introduit dans le chapitre 2. Les fonctionnalités de l'application et l'architecture sont modélisées avec DIPLODOCUS, le profil UML introduit en 2006. Ce dernier est conçu pour la modélisation des systèmes complexes à haut niveau d'abstraction : abstraction des données et des fonctions. DIPLODOCUS est supporté par l'atelier logiciel de modélisation libre TTool qui était doté, préalablement à ce travail, de plusieurs fonctionnalités : la modélisation graphique des diagrammes UML/DIPLODOCUS, une simulation rudimentaire des modèles, et une transformation de modèles automatisée visant les langages formels LOTOS et UPPAAL. Les principales contributions apportées lors de cette thèse concernent le domaine de la simulation,

l'élargissement de la couverture de modèles, l'expression de propriétés fonctionnelles et l'optimisation du flux de conception:

- Une stratégie efficace **de simulation et validation** a été conçue pour compléter les capacités formelles de l'environnement. Le nouvel algorithme de simulation exploite largement les propriétés du modèle d'application en ce qui concerne la granularité et les abstractions. Une sémantique d'exécution pour les opérateurs DIPLODOCUS a été définie, elle est mise à profit dans la simulation et correspond au niveau d'abstraction du profil.

- Un effort s'est porté sur la recherche d'un **compromis entre la couverture limitée offerte par les techniques de simulation conventionnelles et l'exhaustivité de la vérification formelle**. Afin d'offrir un intermédiaire entre ces deux extrêmes, une approche de simulation étendue a été définie et étendue. Notamment, il est proposé un algorithme qui analyse statiquement les applications DIPLODOCUS et identifie l'ensemble des variables d'état qui est significatif à un endroit donné dans l'application. Des méthodes sont aussi présentées pour exploiter les résultats de l'analyse statique lors de la simulation, afin d'examiner plusieurs branches d'exécution. Dans le cas ou des états récurrents du système sont rencontrés, la simulation de certaines branches peut être abandonnée. Contrairement aux techniques de *Model Cheking* conventionnelles, la couverture du modèle d'application est variable. Aussi, les contraintes de l'architecture matérielle sont prises en compte ce qui permet de limiter considérablement le problème d'explosion combinatoire qui est rencontrée lorsque l'on prend en compte l'application uniquement (l'architecture contraint les traces d'exécution).

- La vérification d'un modèle est souvent contrariée par l'obligation de s'appuyer sur des langages de propriétés complètement différents de ceux utilisés pour la conception du système. Par exemple, **les propriétés devant être respectées par un modèle UML devrait être exprimables dans le même langage**, en UML. Pour répondre à ce besoin, un langage graphique d'expression de propriétés nommé TEPE (Temporal Expression Language) est introduit. Il est basé sur les diagrammes paramétriques SysML. TEPE enrichit l'expressivité des autres langages d'expression de propriétés bien établis en particulier avec la notion de temps physique, la facilité d'exprimer des propriétés logiques et temporels, et enfin la possibilité de combiner aisément les différents opérateurs. De plus, grâce à une composition en deux dimensions, TEPE supporte à la fois des formalismes orientés événements et états. Enfin, TEPE est doté d'une sémantique formelle qui fait également partie de la contribution apportée par cette thèse.

- Le flot de conception a été optimisé puisque l'utilisateur n'a **pas besoin de se référer au modèle exécutable pour pouvoir déboger son modèle**. La construction, le débogage et la vérification peuvent ainsi désormais être effectués de manière

transparente dans le même environnement, en utilisant le même langage, sans avoir à écrire une seule ligne de code.

- Enfin, une partie importante de cette thèse porte sur le développement d'un prototype qui **implémente l'ensemble des concepts mentionnés ci-dessus**.

10.4 Plan de la thèse et résultats

Cette thèse est structurée en 8 chapitres principaux:

Le premier chapitre explore les autres travaux visant la vérification des propriétés non fonctionnelles et fonctionnelles des systèmes sur puce (SoC) dès le début du processus de conception. En particulier, la section 3.2 présente des travaux en rapport avec ce travail et avec la sémantique de simulation DIPLODOCUS (section 5.4). Cette section aborde les points communs de toutes les approches de modélisation, à savoir les modèles de calcul (MoC). La section 3.4 porte sur l'exploration de l'espace de conception (DSE) au niveau système, où les modèles sont adaptés à des aspects de performance et ne sont pas nécessairement fonctionnels. Dans ce contexte, les approches peuvent être classées en trois catégories simples auxquelles des sections séparées sont consacrées : les approches formelles / statiques (section 3.4.1), les approches centrées sur la simulation (section 3.4.3) et les variantes hybrides (section 3.4.4). Pour une classification plus approfondie des approches, il convient de se référer à la section 3.3. Les approches formelles et celles basées sur la simulation y sont comparées ainsi que les approches visant l'ensemble des systèmes ou encore celles visant uniquement les architectures de communication (section 3.4.5). Les environnements peuvent être limités à un type particulier d'applications (telles que les applications de traitement du signal) et atteindre une couverture différente de l'espace de conception en fonction de leur méthodologie et du modèle de calcul . La simulation en tant que méthode de vérification est un aspect central de cette thèse ; pour cette raison les méthodes visant à accélérer les techniques de simulation traditionnelles sont détaillées dans la section 3.4.6. La section 3.5 présente les avancées de notre langage d'expression de propriétés TEPE par rapport aux approches courantes dans ce domaine. Afin de souligner l'importance des outils pour l'acceptation d'une méthodologie, TTool est comparé à l'état de l'art des environnements de modélisation UML qui proposent des fonctionnalités de visualisation (voir la section 3.6).

10.4.1 TEPE

La vérification de modèles abstraits au niveau système est toujours entravée par l'expertise nécessaire dans la logique temporelle. Les développeurs moins familiers avec ce domaine apprécieraient certainement un langage de vérification qui corresponde au niveau d'abstraction du modèle à vérifier. Pour répondre à ce besoin, nous préconisons TEPE, un langage graphique et convivial pour l'expression de propriétés temporelles définies

formellement avec le formalisme nommé *fluent*. Grâce à sa définition, TEPE supporte aussi bien le formalisme à base d'états que le formalisme à base de transitions/événements. Ainsi, TEPE s'applique à une grande variété de systèmes définis en termes d'états (attributs) et de transitions entre ces états (signaux).

L'importance croissante des systèmes temps-réel dans les applications critiques a stimulé les travaux de recherche sur les techniques de modélisation qui combinent la convivialité d'UML/SysML avec la formalité de langages de vérification tels que la logique temporelle, UPPAAL [24], etc. Jusqu'à présent , l'utilisation de SysML dans les méthodes de vérification a été entravée par la formalité insuffisante des diagrammes d'exigences et par l'absence de puissants langages d'expression de propriété. Ainsi, les profils UML communs nécessitent l'utilisation de la logique temporelle (par exemple CTL) qui risquent de ne pas correspondre au niveau d'abstraction du modèle du système. La norme MARTE intègre VSL [92], qui vise plutôt la spécification de valeurs liées aux aspects non fonctionnels. Quand il s'agit de la vérification du comportement séquentiel, MARTE propose le Clock Constraint Specification Language (CCSL). Ce cadre théorique est doté d'une base mathématique solide, mais s'en remet au développeur pour l'abstraction des transitions du système en terme d'horloges et pour les questions pratiques relatives aux outils. Les utilisateurs dépourvus de connaissances mathématiques risquent d'être découragés par les méthodes formelles qui ne fournissent pas une visualisation conviviale.

Pour pallier les insuffisances susmentionnées, ce travail enrichit les diagrammes paramétriques SysML avec TEPE, un langage graphique mais formel pour exprimer les propriétés logiques et temporelles. Avec TEPE, les divers éléments de conception, comme les attributs de blocs SysML et les signaux, peuvent être combinés avec la logique (par exemple, la séquence des signaux) et les opérateurs temporels (par exemple, un intervalle de temps pour la réception de signaux) pour composer des propriétés graphiques complexes.

TEPE offre une interface intuitive et une stratégie à deux dimensions pour composer des propriétés de sûreté et de liveness qui reposent sur les contraintes. Le langage est accompagné d'une représentation graphique qui provient des diagrammes paramétriques SysML. Par ailleurs, TEPE pourrait être introduit en SysML d'OMG et en dans une grande variété de dérivés de SysML. Pour en faire la démonstration, nous intégrons TEPE dans deux environnements UML / SysML assez différents par leurs objectifs et leurs sémantiques : AVATAR [1] et DIPLODOCUS. Grâce à l'utilisation conjointe de TEPE et d'un environnement UML / SysML, la capture d'exigences, l'analyse, la conception, les taches de description et de vérification de propriétés peuvent être accomplies de façon transparente dans le même langage avec un seul outil (cf. Chapitre 7) . Pour utiliser TEPE, le développeur doit seulement présenter des compétences mineures en UML et il n'est pas obligé de maîtriser les langages formels tels que CTL ou UPPAAL.

[1] Automated Verification of reAl Time softwARe

Le chapitre est organisé comme suit : la section 4.2 présente les formalismes Metric Temporal Logic et Fluent Temporal Logic qui servent de base formelle pour les contraintes TEPE. La section 4.3 introduit le langage TEPE d'abord d'une manière intuitive et ensuite formellement. Enfin, deux sections sont consacrées à l'intégration de TEPE dans deux profils UML / SysML représentatifs, notamment AVATAR (section 4.4) et DIPLODOCUS (section 4.5). Une étude de cas en DIPLODOCUS et AVATAR est présentée, elle s'étend sur la conception, les étapes de modélisation et de vérification de propriétés d'un four à micro-ondes. Enfin, la section 4.6 conclut ce chapitre.

10.4.2 Simulation

Le chapitre précédent a préconisé un langage formel et graphique pour exprimer les propriétés du système. Pour vérifier ces propriétés, nous devons tout d'abord donner vie aux modèles UML en les rendant exécutables. Alors que la transformation de modèles UML en code exécutable est démontrée dans le chapitre 7, ce chapitre propose une nouvelle approche de simulation pour les modèles à haut niveau d'abstraction des systèmes sur puce. Dans la partie introductive, nous motivons notre décision de ne pas nous appuyer sur la bibliothèque de simulation SystemC, qui est pourtant très usitée dans notre domaine (section 5.3). De plus, le modèle de calcul Discrete Event est analysé du point de vue performance de simulation dans la section 5.2. La section 5.4 révèle la sémantique de simulation de DIPLODOCUS et aborde les hypothèses faites sur l'architecture, l'application et le mapping. Pour mieux expliquer les abstractions comme la séparation d'application et d'architecture, un exemple est présenté avec un bus embarqué CAN dans la section 5.4.4. La section 5.5 donne des précisions sur la stratégie de simulation et familiarise le lecteur avec les phases de simulation et certains mécanismes de synchronisation. Pour plus d'explications sur les problèmes liés à l'implémentation, le lecteur peut se référer à la section 5.6.

Notre approche de simulation rapide est centrée autour de deux principes fondamentaux :

- Une méthodologie de modélisation qui implique l'abstraction de données et de fonctionnalités.
- Une stratégie de simulation qui exploite efficacement les propriétés des modèles de haut niveau. Ainsi, la granularité de la simulation correspond à la granularité du modèle d'application.

Cela implique que la vitesse de simulation dépend essentiellement de la façon dont l'application est modélisée. En effet, plus la granularité du modèle est grande, plus les actions contiguës considérées par le simulateur (les transactions) sont longues et le plus efficace (en termes de cycles / seconde) est la simulation. Si la synchronisation

des processus concurrents ne nécessite pas de tronquer des transactions, le simulateur exécute des transactions couvrant plusieurs cycles d'horloge à la fois.

10.4.3 Couverture

Comme la vérification formelle des modèles de bas niveau est généralement confrontée au problème d'explosion combinatoire, cette vérification est simplement appliquée aux sous-parties du système où les données sont abstraite par leur simple présence ou absence. Cette limitation est écartée en augmentant le niveau d'abstraction. Cependant, il reste un obstacle majeur pour la vérification des modèles de taille moyenne au niveau système. Par ailleurs, il est extrêmement difficile de représenter des modèles de mapping de haut niveau (des modèles comprenant une application et l'architecture) en des langues formelles. Des méthodes telles que DIPLODOCUS ont été proposées pour transformer des modèles graphiques de haut niveau (UML, etc) en une représentation appropriée pour la vérification. Pour cela, des modèles UML doivent être dotés d'une sémantique formelle. Néanmoins, nous avons constaté par expérience que la transformation de modèles sophistiqués de mapping résulte en des structures syntaxiques complexes qui poussent souvent les modèles checkers UPPAAL [24] et CADP [38] à leurs limites.

Toutefois, même si le model checker est capable de gérer la spécification, la couverture de l'espace d'états ne peut pas être variée : la vérification formelle est exhaustive par définition et par conséquent les model checkers sont conçus pour explorer l'espace d'états exhaustivement. Pour cette raison, afin de varier la couverture du modèle source, il est nécessaire de reconfigurer l'algorithme de transformation et de régénérer le modèle formel. Cependant, du point de vue d'un développeur, il serait souhaitable de générer un seul modèle exécutable pour ensuite analyser différentes parties intéressantes de son espace d'état.
L'espace de conception peut être réduit à l'aide de critères de couverture conventionnelle (les branches couvertes, des commandes, des tâches, des conditions, etc), des connaissances d'expert fournis par l'utilisateur (par exemple les parties potentiellement critiques du flux de contrôle), ou des heuristiques tenant compte de propriétés (non-) fonctionnelles. L'objectif est de permettre la prise dynamique de décisions, au temps d'exécution de la simulation. De cette façon, le développeur peut trouver plus facilement un compromis entre l'efficacité de simulation et la couverture sans régénérer le modèle formel. Le point de fonctionnement varie ainsi entre les deux cas extrêmes de la vérification formelle et de la simulation, ce qui met en évidence le curseur dans la figure 6.1). Bien entendu, le fait de s'éloigner d'une vérification formelle exhaustive ne permet plus de fournir le même degré de confiance et garanties.

Pour répondre à ce besoin de compromis, nous proposons une nouvelle façon d'améliorer la couverture de simulation des modèles de haut niveau basé sur le model checking et

de techniques d'analyse statique. Un simulateur particulièrement adapté aux modèles de haut niveau de SoC a été introduit dans le chapitre 5. Il permet au développeur d'explorer manuellement plusieurs branches en sauvegardant l'état de la simulation à un point de décision et de le reprendre plus tard. Au stade qui est décrit dans le chapitre 5 le simulateur manque de fonctionnalités de stockage d'état et des techniques de comparaison qui sont essentielles pour la fusion de chemins d'exécution logiquement équivalentes. Ce chapitre présente des solutions à ces défauts. Des modèles de mapping DIPLODOCUS sont statiquement analysés pour repérer les éléments ne faisant pas partie du vecteur d'états afin d'accélérer l'identification des états du système récurrents. Section 6.3 donne des précisions sur cette procédure. La section 6.4 démontre une façon pour localiser les points dans le modèle où les exécutions sont susceptibles d'être fusionnées, et donc les états doivent être comparés. La section 6.5 donne une idée au lecteur sur l'implémentation des relations de dépendances et le hashage d'états. Des méthodes pour l'identification des états récurrents du système sont abordées dans la section 6.5.5.

10.4.4 Tooling

Jusqu'ici, les contributions de ce travail ont été présentées principalement d'un point de vue méthodologique, décrivant les idées de base et la théorie sous-jacente. Même si quelques indications sur les questions d'implémentation ont été fournies à la fin de chaque chapitre, il reste encore à mettre tous ces éléments dans le contexte de la chaîne de compilation. L'objectif de ce chapitre est de compléter ce travail en examinant les contributions qui ont été exposées précédemment dans le cadre global de TTool, d'un point de vue pratique. La section 7.2 aborde la structure interne des éléments de l'ensemble d'outillage, notamment TTool et le moteur de simulation. La section 7.3 décrit l'étape de transformation du modèle graphique UML en code C++. Les fonctionnalités de simulation interactive et leur importance pour le débogage sont présentées dans la section 7.4. Ainsi, une attention particulière est attirée sur l'interaction entre le simulateur et TTool, qui ont été développés dans différents langages de programmation.

10.4.5 Evaluation

Jusqu'ici, nous avons proposé le langage de spécification de propriétés TEPE, un environnement de simulation pour les modèles de haut niveau des systèmes sur puce, une méthodologie pour améliorer la couverture de ce dernier et enfin une suite d'outils qui implémente entièrement le flot de conception. Nos efforts ont été motivés par les inconvénients des environnements existants (identifiés dans le chapitre 3) par rapport au niveau d'abstraction, à la vitesse de simulation, à la représentation des flux de contrôle et à l'indéterminisme. Bien que les preuves de l'applicabilité de plusieurs concepts aient été fournies tout au long de cette thèse, ce chapitre met l'accent sur un exemple plus complet. Il est inspiré du domaine des télécommunications et il met en évidence le modèle d'un récepteur pour la norme 802.11p, ses propriétés ainsi que sa simulation et

sa vérification. Plus précisément, l'objectif de ce chapitre est :

- d'illustrer le concept de modélisation DIPLODOCUS et de ses résultats avec un système embarqué concret et existant ;
- démontrer que TEPE est capable d'exprimer des propriétés pertinentes des SoC ;
- d'illustrer comment les résultats de simulation peuvent être utilisés ;
- de fournir des preuves pour l'utilité d'une couverture élargie de simulation ;
- d'illustrer l'interaction des composants de l'outillage.

Cependant, il reste hors de portée de cette thèse de confronter les résultats de simulation avec des mesures de performances effectuées sur le matériel réel. En ce qui concerne des propriétés non fonctionnelles, la comparaison publiée dans [55] nous rend confiants que la méthodologie de DIPLODOCUS permet d'obtenir des chiffres de performances réalistes. Le développeur est en effet assisté dans son effort d'identifier une implémentation qui est conforme aux contraintes (non-) fonctionnelles. Concernant les propriétés fonctionnelles, un raffinement des modèles DIPLODOCUS en préservant ces derniers est actuellement étudié dans le cadre d'une thèse de doctorat [84].

Ce chapitre est structuré comme suit : la section 8.2 expose le domaine d'application et le système à modéliser et définit des notions connexes. La section 8.2.2 présente les modèles DIPLODOCUS de l'application et de l'architecture qui seront exploités pour obtenir des chiffres de performance et pour vérifier la conformité à des propriétés dans la section 8.3. La dernière section analyse également les performances du simulateur, à la fois en mode de simulation classique et en mode de couverture améliorée. La section 8.4 conclut finalement ce chapitre.

10.4.6 Conclusion

Dans l'introduction, nous avons identifié le besoin d'environnements de modélisation qui décrivent les systèmes embarqués et les systèmes sur puce à un haut niveau d'abstraction. De cette manière, le développeur peut obtenir un premier aperçu du comportement du système en cours de conception, même si les modèles de bas niveau ne sont pas disponibles ou trop coûteux à construire. De même, il pourrait être intéressant de faire l'abstraction de modèles détaillés pour accélérer la simulation ou étendre sa portée (par exemple du niveau module au niveau système). Les abstractions sont essentielles pour l'exploration de l'espace de conception, où un grand nombre d'alternatives d'implémentations doit être analysé. Dans le chapitre 3, nous avons examiné quelques exemples représentatifs des nombreux travaux de même nature. La catégorisation des travaux a révélé la différence fondamentale des modèles statiques et dynamiques, ces derniers nécessitant une représentation implicite ou explicite d'états de système. La modélisation de comportement dans les diagrammes DIPLODOCUS est basée sur une

représentation d'états et donc (en général) engendre une traversée de l'espace d'états à être vérifiée. Cette thèse est consacrée à l'exécution et la vérification des modèles de haut niveau qui présentent une sémantique similaire à celle de DIPLODOCUS. La section 9.1 résume les contributions de cette thèse. Dans la section 9.2 nous récapitulons les inconvénients des travaux liés identifiés dans la section 3.7 et nos affirmations initiales. Nous résumons les avantages de DIPLODOCUS en général, et de ce travail en particulier, et nous les associons aux exemples fournis tout au long de cette thèse. Enfin, la section 9.3 présente des travaux futurs qui sont susceptibles de compléter les solutions présentées.

References

[1] CoFluent Studio www.cofluentdesign.com. 39

[2] Coware Virtual Platforms www.coware.com. 39

[3] SysML companion. In *http://www.realtimeatwork.com/?page_id=683*. 47

[4] Vast System Engineering Tools www.vastsystems.com. 39

[5] Accellera Organization Inc. SystemVerilog 3.1a Language Reference Manual, www.systemverilog.org. 45

[6] Accellera Organization Inc. Property specification language, reference manual, version 1.1. 2004. 45

[7] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Pearson Education, Boston, MA, USA, 2007. 112, 113, 163

[8] Solange Ahumada, Ludovic Apvrille, Tomás Barros, Antonio Cansado, Eric Madelaine, and Emil Salageanu. Specifying fractal and GCM components with UML. In *Proceedings of the XXVI International Conference of the Chilean Society of Computer Science*, pages 53–62, Washington, DC, USA, 2007. IEEE Computer Society. 17

[9] Charles André, Frédéric Mallet, and Robert de Simone. Modeling time(s). In *MoDELS*, pages 559–573, 2007. 46

[10] F. Angiolini, J. Ceng, R. Leupers, F. Ferrari, C. Ferri, and L. Benini. An integrated open framework for heterogeneous MPSoC design space exploration. *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, **1**:1–6, March 2006. 8, 38

[11] L. Apvrille. TTool for DIPLODOCUS: An Environment for Design Space Exploration. In *Proceedings of the 8th Annual International Conference on New Technologies of Distributed Systems (NOTERE'2008)*, Lyon, France, June 2008. 17, 21

[12] L. Apvrille and A. Becoulet. Fast and multi-platform prototyping of embedded systems from UML/SysML models. In *Proceedings of the 14th Sophia Antipolis Mircroelectronics Forum (SAME 2011)*, October 2011. 68

[13] L. Apvrille, J.-P. Courtiat, C. Lohr, and P. de Saqui-Sannes. TURTLE: A real-time UML profile supported by a formal validation toolkit. In *IEEE transactions on Software Engineering*, **30**, pages 473–487, Jul 2004. 17, 47

[14] L. Apvrille, W. Muhammad, R. Ameur-Boulifa, S. Coudert, and R. Pacalet. A UML-based environment for system Design Space Exploration. *Electronics, Circuits and Systems, 2006. ICECS '06. 13th IEEE International Conference on*, pages 1272–1275, Dec. 2006. 7, 21, 47

[15] Ludovic Apvrille, Ahlem Mifdaoui, and Pierre De Saqui-Sannes. Real-time distributed systems dimensioning and validation: The TURTLE method. *Studia Informatica Universalis*, **8, no 3**:47–69, October 2010. 17, 31

[16] Ludovic Apvrille and Pierre De Saqui-Sannes. Making formal verification amenable to real-time UML practitioners. In *Proceedings of the 12th European Workshop on Dependable Computing*, Toulouse, France, May 2009. 7, 17, 128

[17] Tero Arpinen, Erno Salminen, Timo Hämäläinen, and Marko Hännikäinen. Performance evaluation of UML2-modeled embedded streaming applications with system-level simulation. *EURASIP Journal on Embedded Systems*, **2009**, March 2009. 39

[18] Artisan. Artisan studio, www.artisansoftwaretools.com/products/artisan-studio. 2009. 48

[19] M. Audrain and B. Marconato. Topcased 3.4 tutorial - requirement management. In *http://www.topcased.org/index.php? documentsSynthesis=y&Itemid=59*, 2010. 47

[20] Brian Bailey, Grant Martin, and Andrew Piziali, 2007. 35

[21] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: an integrated electronic system design environment. *Computer*, **36**[4]:45–52, April 2003. 39

[22] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in BIP. pages 3–12, Sept. 2006. 34

[23] Sciuto Beltrame, Fossati. A real-time application design methodology for MPSoCs. *Design, Automation and Test in Europe, 2009. DATE '09. Proceedings*, **1**, March 2009. 44

[24] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In *Lecture Notes on Concurrency and Petri Nets*. W. Reisig and G. Rozenberg (eds.), LNCS 3098, Springer-Verlag, 2004. 51, 110, 171, 173

[25] Tobias Blickle, Tobias Blickle, Jurgen Teich, and Lothar Thiele. System-level synthesis using evolutionary. *Design Automation for Embedded Systems*, **3**:23–58, 1998. 9

[26] A. Bobrek, J.J. Pieper, J.E. Nelson, J.M. Paul, and D.E. Thomas. Modeling shared resource contention using a hybrid simulation/analytical approach. *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, **2**:1144–1149 Vol.2, Feb. 2004. 41

[27] J. P. Calvez and D. Isidoro. A codesign experience with the mcse methodology. In *Proc. Third International Workshop on Hardware/Software Codesign*, pages 140–147, September 22–24, 1994. 39

[28] S. Chakraborty, S. Kunzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 190–195, 2003. 31

[29] Zhenqiang Chen, Baowen Xu, and Jianjun Zhao. An overview of methods for dependence analysis of concurrent programs. *SIGPLAN Not.*, **37**:45–52, August 2002. 117

[30] Bruce A. Cota and Robert G. Sargent. A modification of the process interaction world view. *ACM Trans. Model. Comput. Simul.*, **2**:109–129, April 1992. 81

[31] David Currie, Xiushan Feng, Masahiro Fujita, Alan J. Hu, Mark Kwan, and Sreeranga Rajan. Embedded software verification using symbolic execution and uninterpreted functions. *Int. J. Parallel Program.*, **34**[1]:61–91, 2006. 33

[32] E. C. da Silva and E. Villani. Integrando SysML e model checking para v&v de software crítico espacial. In *Brasilian Symposium on Aerospace Engineering and Applications, São José dos Campos, SP, Brasil, in Portuguese*, September 2009. 47

[33] Werner Damm and David Harel. Lscs: Breathing life into message sequence charts. *Formal Methods in System Design*, **19**[1]:45–80, 2001. 46

[34] Peter J. Denning and Jeffrey P. Buzen. The operational analysis of queueing network models. *ACM Comput. Surv.*, **10**[3]:225–261, 1978. 27, 31

[35] S. Edwards, L. Lavagno, E.A. Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: formal models, validation, and synthesis. *Proceedings of the IEEE*, **85**[3]:366–390, Mar 1997. 1, 2, 22, 165, 166

[36] J. Eker, J.W. Janneck, E.A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Yuhong Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, **91**[1]:127–144, Jan 2003. 34

[37] George S. Fishman. *Discrete-Event Simulation: Modeling, Programming, and Analysis*. Springer-Verlag, Berlin, 2001. 81

[38] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In *Computer Aided Verification (CAV'2007)*, **4590**, pages 158–163, Berlin Germany, 2007. 18, 110, 173

[39] Patrice Godefroid and Sarfraz Khurshid. Exploring very large state spaces using genetic algorithms. In *TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 266–280, London, UK, 2002. Springer-Verlag. 162

[40] Matthias Gries. Methods for evaluating and covering the design space during early design development. *Integration, the VLSI Journal*, **38**[2]:131 – 183, 2004. 1, 8, 166

[41] Thorsten Grotker. *System Design with SystemC*. Kluwer Academic Publishers, Norwell, MA, USA, 2002. 36

[42] Arne Hamann, Marek Jersak, Kai Richter, and Rolf Ernst. A framework for modular analysis and exploration of heterogeneous embedded systems. *Real-Time Syst.*, **33**[1-3]:101–137, 2006. 9, 31

[43] D. Harel and R. Marelly. Playing with time: On the specification and execution of time-enriched LSCs. In *MASCOTS '02: Proceedings of the 10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, page 193, Washington, DC, USA, 2002. IEEE Computer Society. 46

[44] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, **8**[3]:231–274, 1987. 25

[45] C. Haubelt, T. Schlichter, J. Keinert, and M. Meredith. SystemCoDesigner: Automatic Design Space Exploration and rapid prototyping from behavioral models. In *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pages 580–585, June 2008. 39

[46] M. Hause and J. Holt. Testing solutions with UML/SysML. In *http://www.artist-embedded.org/docs/Events/2010/UML_AADL /slides/Session1_Matthew_Hause.pdf*, 2010. 47

[47] M. Hendriks and M. Verhoef. Timed automata based analysis of embedded system architectures. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 8 pp.–, April 2006. 33

[48] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis - the SymTA/S approach. *Computers and Digital Techniques, IEE Proceedings -*, **152**[2]:148–166, Mar 2005. 31

[49] Thomas A. Henzinger and Joseph Sifakis. The discipline of embedded systems design. *Computer*, **40**:32–40, 2007. 1, 165

[50] Paula Herber, Joachim Fellmuth, , and Sabine Glesner. Model checking SystemC designs using timed automata. In *Proceedings of the 6th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. ACM, 2008. 33

[51] Paula Herber, Florian Friedemann, and Sabine Glesner. Combining model checking and testing in a continuous hw/sw co-verification process. In Catherine Dubois, editor, *3rd International Conference on Tests and Proofs (TAP'09)*, **LNCS**, pages 121–136. Springer, 2009. 33

[52] Gerard J. Holzmann. State compression in spin: Recursive indexing and compression training runs. In *Proceedings of the third international Spin Workshop*, 1997. 162

[53] G.J. Holzmann. The model checker spin. *Software Engineering, IEEE Transactions on*, **23**[5]:279–295, May 1997. 111

[54] L.S. Indrusiak, A. Thuy, and M. Glesner. Executable system-level specification models containing UML-based behavioral patterns. *Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE '07*, pages 1–6, April 2007. 37

[55] Chafic Jaber. High-level SoC modeling and performance estimation applied to a multi-core implementation of LTE eNodeB physical layer, PHD thesis. September 2011. 13, 85, 87, 139, 175

[56] Bob Jenkins. A hash function for hash table lookup, www.burtleburtle.net/bob/hash/doobs.html. 2006. 125

[57] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing '74: Proceedings of the IFIP Congress*, pages 471–475. North-Holland, New York, NY, 1974. 25

[58] Kangas. Methods and implementations for automated System on Chip architecture exploration. *PHD Thesis*, September 2006. 39

[59] Terence Kelly, Kai Shen, Alex Zhang, and Christopher Stewart. Operational analysis of parallel servers. In *MASCOTS*, pages 227–236, 2008. 31

[60] Bart Kienhuis, Ed F. Deprettere, Pieter van der Wolf, and Kees A. 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, London, UK, UK, 2002. Springer-Verlag. 3, 8, 9, 168

[61] Sungchan Kim and Soonhoi Ha. Efficient exploration of bus-based system-on-chip architectures. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, **14**[7]:681–692, July 2006. 8, 42

[62] D. Knorreck, L. Apvrille, and R. Pacalet. Formal system-level Design Space Exploration. In *Proceedings of the 10th Annual International Conference on New Technologies of Distributed Systems (NOTERE'2010)*, Tozeur, Tunisia, May 2010. 18, 23

[63] Daniel Knorreck, Ludovic Apvrille, and Renaud Pacalet. Fast simulation techniques for Design Space Exploration. In *Objects, Components, Models and Patterns*, **33** of *Lecture Notes in Business Information Processing*, pages 308–327. Springer Berlin Heidelberg, 2009. 18

[64] Daniel Knorreck, Ludovic Apvrille, and Renaud Pacalet. An interactive system level simulation environment for Systems on Chip. In *ERTSS - Embedded Real Time Software and Systems*, May 2010. 18

[65] Daniel Knorreck, Ludovic Apvrille, and Pierre De Saqui-Sannes. TEPE: A SysML language for timed-constrained property modeling and formal verification. In *Proceedings of the UML&Formal Methods Workshop (UML&FM)*, Shanghai, China, November 2010. 17, 47

[66] Alfred Koelbl and Carl Pixley. Constructing efficient formal models from high-level descriptions using symbolic simulation. *International Journal of Parallel Programming*, **33**[6]:645–666, December 2005. 32

[67] Ron Koymans. *Specifying Message Passing and Time-Critical Systems with Temporal Logic*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1992. 52

[68] Kraemer S., Gao L., Weinstock J., Leupers R. Ascheid G. and H. Meyr. HySim: A Fast Simulation Framework for Embedded Software Development. In *Proceedings of the 5th Conference on Hardware/Software Codesign (CODES+ISSS '07) and System Synthesis*, Salzburg, Austria, 2007. 44

[69] J. Kreku, T. Kauppi, and J.-P. Soininen. Evaluation of platform architecture performance using abstract instruction-level workload models. In *System-on-Chip, 2004. Proceedings. 2004 International Symposium on*, pages 43–48, Nov. 2004. 40

[70] Jari Kreku, Mika Hoppari, Tuomo Kestilä, Yang Qu, Juha-Pekka Soininen, Per Andersson, and Kari Tiensyrjä. Combining UML2 application and SystemC platform modelling for performance evaluation of real-time embedded systems. *EURASIP J. Embedded Syst.*, **2008**:1–18, 2008. 15, 38

[71] S. Kunzli, F. Poletti, L. Benini, and L. Thiele. Combining simulation and formal methods for system-level performance analysis. *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, **1**:1–6, March 2006. 41

[72] K. Lahiri, A. Raghunathan, and S. Dey. System-level performance analysis for designing on-chip communication architectures. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, **20**[6]:768–783, Jun 2001. 8, 42

[73] K. Lahiri, A. Raghunathan, and S. Dey. Design Space Exploration for optimizing on-chip communication architectures. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, **23**[6]:952–961, June 2004. 8, 42

[74] Edward A. Lee and Alberto Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **17**:1217–1229, 1998. 22, 35

[75] Emmanuel Letier, Jeff Kramer, Jeff Magee, and Sebastian Uchitel. Fluent temporal logic for discrete-time event-based models. In *Proceedings of the 10th European software engineering conference*, ESEC/FSE-13, pages 70–79, New York, NY, USA, 2005. ACM. 53, 54

[76] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *DAC '95: Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*, pages 456–461, New York, NY, USA, 1995. ACM. 32

[77] P. Lieverse, P. van der Wolf, E. Deprettere, and K. Vissers. A methodology for architecture exploration of heterogeneous signal processing systems. In *Signal Processing Systems, 1999. SiPS 99. 1999 IEEE Workshop on*, pages 181–190, 1999. 2, 40, 166

[78] Yi-Bing Lin and P.A. Fishwick. Asynchronous parallel discrete event simulation. *Systems, Man and Cybernetics, Part A, IEEE Transactions on*, **26**[4]:397–412, Jul 1996. 26

[79] Gabor Madl, Nikil Dutt, and Sherif Abdelwahed. Performance estimation of distributed real-time embedded systems by discrete event simulations. In *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 183–192, New York, NY, USA, 2007. ACM. 33

[80] Frédéric Mallet. Clock constraint specification language: specifying clock constraints with UML/MARTE. *Innovations in Systems and Software Engineering*, **4**[3]:309–314, October 2008. 46

[81] Radu Marculescu, Umit Y. Ogras, and Nicholas H. Zamora. Computation and communication refinement for multiprocessor SoC design: A system-level perspective. *ACM Trans. Des. Autom. Electron. Syst.*, **11**[3]:564–592, 2006. 31

[82] Mokhoo Mbobi and Frederic Boulanger. An overall specification of a meta-model of computation for model-driven embedded system modeling. In *CTS '06: Proceedings of the International Symposium on Collaborative Technologies and Systems*, pages 194–199, Washington, DC, USA, 2006. IEEE Computer Society. 23, 35

[83] J. Mitola. The software radio architecture. *Communications Magazine, IEEE*, **33**[5]:26 –38, may 1995. 140

[84] Hocine Mokrani. Design and formal validation of embedded systems, phd thesis. to appear in 2012. 139, 175

[85] Wolfgang Mueller, Juergen Ruf, Dirk Hoffmann, Joachim Gerlach, Thomas Kropf, and Wolfgang Rosenstiehl. The simulation semantics of SystemC. In *In Proc. of DATE 2001. IEEE CS*, pages 64–70. Press, 2001. 82

[86] N.-u.-I. Muhammad, R. Rasheed, R. Pacalet, R. Knopp, and K. Khalfallah. Flexible baseband architectures for future wireless systems. In *Digital System Design Architectures, Methods and Tools, 2008. DSD '08. 11th EUROMICRO Conference on*, pages 39 –46, sept. 2008. 140

[87] Y.N. Naguib and R.S. Guindi. Speeding up SystemC simulation through process splitting. In *Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE '07*, pages 1–6, April 2007. 44, 82

[88] Mangala Gowri Nanda and S. Ramesh. Slicing concurrent programs. In *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '00, pages 180–190, New York, NY, USA, 2000. ACM. 117

[89] Iulian Ober and Iulia Dragomir. OMEGA2: A new version of the profile and the tools (regular paper). In *UML&AADL'2009 - 14th IEEE International Conference on Engineering of Complex Computer Systems*, pages 373–378, Potsdam, June 2009. IEEE. 47

[90] Members of the SystemC Verification Working Group. SystemC Verification Standard Specification Version 1.0e, www.systemc.org. 2003. 46

[91] OMG. UML 2.0 superstructure specification. In *http://www.omg.org/docs/ptc/03-08-02.pd*, Geneva, 2003. 7, 47

[92] OMG. A UML profile for MARTE, beta 2, www.omg.org. 2008. 46, 51, 171

[93] OMG. A UML profile for MARTE: Modeling and analysis of real-time embedded systems. In *http://www.omgmarte.org/Documents/Specifications/08-06-09.pdf*, 2008. 7, 14

[94] S. Pasricha, Nikil Dutt, and M. Ben-Romdhane. Extending the transaction level modeling approach for fast communication architecture exploration. *Design Automation Conference, 2004. Proceedings. 41st*, pages 113–118, 2004. 8, 42

[95] G. Pedroza, L. Apvrille, and D. Knorreck. AVATAR: A SysML environment for the formal verification of safety and security properties. In *The 11th IEEE Conference on Distributed Systems and New Technologies (NOTERÉ2011)*, Paris, France, May 2011. 67

[96] Simon Perathoner, Kai Lampka, and Lothar Thiele. Composing heterogeneous components for system-wide performance analysis. In *Proceedings of Design, Automation and Test in Europe, 2011 (DATE 11)*, Grenoble, France, 2011. 41

[97] D. G. Perez, G. Mouchard, and O. Temam. A new optimized implementation of the SystemC engine using acyclic scheduling. In *Proc. Design, Automation and Test in Europe Conference and Exhibition*, **1**, pages 552–557, February 16–20, 2004. 43, 44, 82

[98] Carl Adam Petri. Fundamentals of a theory of asynchronous information flow. *IFIP Congress 62*, pages 386–390, 1962. 25

[99] Joshua J. Pieper, Alain Mellan, JoAnn M. Paul, Donald E. Thomas, and Faraydon Karim. High level cache simulation for heterogeneous multiprocessors. In *DAC '04: Proceedings of the 41st annual Design Automation Conference*, pages 287–292, New York, NY, USA, 2004. ACM. 8

[100] A. D. Pimentel, S. Polstra, and F. Terpstra. Towards efficient Design Space Exploration of heterogeneous embedded media systems. In *In Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation*, pages 57–73. Springer, LNCS, 2002. 40

[101] A.D. Pimentel, C. Erbas, and S. Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *Computers, IEEE Transactions on*, **55**[2]:99–112, Feb. 2006. 40

[102] A.D. Pimentel, L.O. Hertzbetger, P. Lieverse, P. van der Wolf, and E.E. Deprettere. Exploring embedded-systems architectures with artemis. *Computer*, **34**[11]:57–63, Nov 2001. 40

[103] Andy D. Pimentel, Mark Thompson, Simon Polstra, and Cagkan Erbas. Calibration of abstract performance models for system-level Design Space Eexploration. *J. Signal Process. Syst.*, **50**:99–114, February 2008. 15

[104] Evita Project. Evita, www.evita-project.org. 2011. 87

[105] V. Reyes, W. Kruijtzer, T. Bautista, G. Alkadi, and A. Nunez. A unified system-level modeling and simulation environment for MPSoC design: Mpeg-4 decoder case study. *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, **1**:1–6, March 2006. 38

[106] Rhapsody. Rhapsody, www-01.ibm.com/software/awdtools/rhapsody. 2009. 48

[107] Bastian Ristau, Torsten Limberg, and Gerhard Fettweis. A mapping framework for guided Design Space Exploration of heterogeneous MP-SoCs. *Design, Automation and Test in Europe, 2008. DATE '08*, pages 780–783, March 2008. 39

[108] Christer Sandberg, Andreas Ermedahl, Jan Gustafsson, and Björn Lisper. Faster WCET flow analysis by program slicing. *SIGPLAN Not.*, **41**[7]:103–112, 2006. 32

[109] A. Sangiovanni-Vincentelli. Quo vadis, SLD? reasoning about the trends and challenges of system level design. *Proceedings of the IEEE*, **95**[3]:467–506, March 2007. 1, 23, 165

[110] Tim Schattkowsky, Wolfgang Mueller, and Achim Rettberg. A model-based approach for executable specifications on reconfigurable hardware. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 692–697, Washington, DC, USA, 2005. IEEE Computer Society. 38

[111] Carina Schmidt Knorreck, Raymond Knopp, and Renaud Pacalet. Hardware optimized sample rate conversion for software defined radio. *FREQUENZ, Journal of RF-Engineering and Telecommunications, November-December 2010, Vol 64, No 11-12*, 2010. 141

[112] J. Schnerr, O. Bringmann, A. Viehl, and W. Rosenstiel. High-performance timing simulation of embedded software. pages 290–295, June 2008. 44

[113] Kai Shen, Alex Zhang, Terence Kelly, and Christopher Stewart. Operational analysis of processor speed scaling. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 179–181, New York, NY, USA, 2008. ACM. 31

[114] M. Silbermintz, A. Sahar, L. Peled, M. Anschel, E. Watralov, H. Miller, and E. Weisberger. Soc modeling methodology for architectural exploration and software development. *Electronics, Circuits and Systems, 2004. ICECS 2004. Proceedings of the 2004 11th IEEE International Conference on*, pages 383–386, Dec. 2004. 41

[115] Margaret H. Smith. Events and constraints: a graphical editor for capturing logic properties of programs. In *Proceedings of the 5th International Symposium on Requirements Engineering*, pages 14–22, 2001. 46

[116] Stefan Stattelmann, Oliver Bringmann, and Wolfgang Rosenstiel. Fast and accurate resource conflict simulation for performance analysis of multi-core systems. *Design, Automation and Test in Europe, 2011. DATE '11. Proceedings*, **1**:1–6, March 2011. 15, 43

[117] P. van Stralen and A. D Pimentel. A trace-based scenario database for high-level simulation of multimedia MP-SoCs. In *Proceedings of Int. Conference on Embedded Computer Systems: Architectures, MOdeling and Simulation (SAMOS '10)*, pages 11–19, July 2010. 40

[118] G. Edward Suh, Srinivas Devadas, and Larry Rudolph. Analytical cache models with applications to cache partitioning. In *Proceedings of the 15th international conference on Supercomputing*, ICS '01, pages 1–12, New York, NY, USA, 2001. ACM. 8

[119] Tau. Tau, www-01.ibm.com/software/awdtools/tau. 2009. 48

[120] Topcased. Topcased, www.topcased.org. 2009. 48

[121] Yves Vanderperren and Wim Dehaene. From UML/SysML to matlab/simulink: current state and future perspectives. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 93–93, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association. 47

[122] Verisity Design Inc. e Language Reference Manual, www.ieee1647.org/downloads/prelim_e_lrm.pdf. 2002. 45

[123] E. Viaud, F. Pecheux, and A. Greiner. An efficient TLM/T modeling and simulation environment based on conservative parallel discrete event principles. *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, **1**:1–6, March 2006. 43, 82

[124] Jorgiano Vidal, Florent de Lamotte, Guy Gogniat, Philippe Soulard, and Jean-Philippe Diguet. A co-design approach for embedded system modeling and code generation with UML and MARTE. In *Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE '09.*, pages 226–231, April 2009. 38

[125] A. Viehl, T. Schonwald, O. Bringmann, and W. Rosenstiel. Formal performance analysis and simulation of UML/SysML models for esl design. *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, **1**:1–6, March 2006. 32

[126] Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model checking programs. In *ASE 2000: Proceedings of the 15th IEEE international conference on Automated software engineering*, page 3, Washington, DC, USA, 2000. IEEE Computer Society. 33

[127] Vojin Živojnovic and Heinrich Meyr. Compiled hw/sw co-simulation. In *DAC '96: Proceedings of the 33rd annual Design Automation Conference*, pages 690–695, New York, NY, USA, 1996. ACM. 44

[128] Z. Wang, K. Lu, and A. Herkersdorf. An approach to improve accuracy of source-level TLMs of embedded software. *Design, Automation and Test in Europe, 2011. DATE '11. Proceedings*, **1**:1–6, March 2011. 15, 44

[129] M. Waseem, L. Apvrille, R. Ameur-Boulifa, S. Coudert, and R. Pacalet. Abstract application modeling for system Design Space Exploration. *Digital System Design: Architectures, Methods and Tools, 2006. DSD 2006. 9th EUROMICRO Conference on*, pages 331–337, 0-0 2006. 7

[130] A. Wieferink, M. Doerper, R. Leupers, G. Ascheid, H. Meyr, T. Kogel, G. Braun, and A. Nohl. System level processor/communication co-exploration methodology for multiprocessor system-on-chip platforms. *Computers and Digital Techniques, IEE Proceedings -*, **152**[1]:3–11, Jan. 2005. 8, 38

[131] T. Wild, A. Herkersdorf, and R. Ohlendorf. Performance evaluation for system-on-chip architectures using trace-based transaction level simulation. **1**, pages 1–6, March 2006. 40

[132] Murray Woodside, Dorina C. Petriu, Dorin B. Petriu, Hui Shen, Toqeer Israr, and Jose Merseguer. Performance by unified model analysis (puma). In *WOSP '05: Proceedings of the 5th international workshop on Software and performance*, pages 1–12, New York, NY, USA, 2005. ACM. 34

[133] Xinping Zhu and Sharad Malik. A hierarchical modeling framework for on-chip communication architectures of multiprocessing SoCs. *ACM Trans. Des. Autom. Electron. Syst.*, **12**[1]:6, 2007. 8, 42