Habilitation à Diriger les Recherches

Université de Nice Sophia-Antipolis

Model-Based Design of Complex Embedded Systems

Written by

Ludovic APVRILLE

Enseignant/chercheur
Institut Mines-Telecom, Telecom ParisTech, CNRS LTCI

Defense held on the 29th of November, 2012

Reviewers:

Prof. Jean-Michel Bruel	Université de Toulouse
Prof. Isabelle Chrisment	Université Nancy I
Prof. Frédéric Pétrot	INP Grenoble

Examiners:

Dr. HDR Emmanuelle Encrenaz	Université Paris VI
Directeur de recherche, HDR, Robert de Simone	INRIA Sophia-Antipolis
Prof. Refik Molva	Eurecom

The best books... are those that tell you what you know already.

George Orwell

Remerciements (*acknowledgements*)

Cette section est la seule en français de ce document : l'anglais a été utilisé dans le reste de ce document afin de le rendre plus facilement accessible à mon entourage international de recherche.

J'aimerais remercier avant tout les rapporteurs de mon habilitation : Jean-Michel Bruel, Isabelle Chrisment, Frédéric Pétrot. Ils m'ont fait des remarques judicieuses qui m'ont permis d'améliorer ce manuscrit. J'aimerais aussi remercier Emmanuelle Encrenaz, Refik Molva et Robert de Simone qui me font l'honneur de participer eux aussi au jury. Robert, tu m'as mis sur la voie de cette habilitation, tu m'as guidé vers les bonnes portes pour y parvenir, merci pour cela.
Mes travaux de recherche se sont réalisés au sein d'équipes compétentes et chaleureuses. Tout d'abord, mon doctorat que j'ai réalisé au sein du département Mathématique et Informatique de l'ENSICA-ISAE, sous la direction de Patrick Sénac et de Michel Diaz : ils m'ont appris le métier de chercheur dans une ambiance studieuse et amicale. Mon post-doctorat s'est effectué dans l'équipe de recherche de Ferhat Khendek à l'université Concordia sur des problématiques qui m'ont permis d'étendre mon champ de vision. Enfin, mon travail actuel à Telecom ParisTech qui m'offre un cadre très favorable pour développer mes idées. En particulier, Bruno Thédrez, responsable du département, et Renaud Pacalet, responsable scientifique de l'équipe du LabSoC, sont à l'écoute et prêts à m'inciter à l'aller de l'avant. Mes collègues du LabSoC sont toujours très ouverts aux échanges, et me permettent ainsi de confronter mes travaux aux leurs, toujours avec le souci de jongler entre problématiques fondamentales et applicabilité à des problèmes plus concrets. Mes remerciements vont également à tous les thésards que j'ai (co-)encadrés ou que j'encadre : ils participent bien entendu au succès de la recherche du LabSoC.
Les collaborations avec d'autres laboratoires académiques ont aussi beaucoup contribué à mes résultats : les départements Réseau et Sécurité et Multimédia d'Eurecom, le LIP6 et l'ISAE. J'aimerais particulièrement remercier Pierre de Saqui-Sannes, qui a su me m'épauler depuis ma thèse jusqu'à cette habilitation, et qui est l'un des plus fervents supporters de l'outil TTool : j'espère que nous continuerons encore longtemps à collaborer. Je remercie aussi les partenaires industriels pourvoyeurs d'études de cas garantes de l'applicabilité de mes travaux, et entre autre Texas Instruments, Freescale, et les membres du projet européen EVITA.
Le personnel de Télécom ParisTech et d'Eurecom m'a apporté une précieuse aide au quotidien: je les remercie vivement pour leur réactivité et leur compétence.
Le pilier cental de ma vie reste ma famille, Axelle ma femme et Guillaume mon fils, qui m'apportent joie et bonheur au quotidien, sans bien sûr oublier Pico, la mascotte de la famille. Merci Axelle pour tous les échanges fructueux que nous avons au jour le jour, en particulier sur le thème de la sécurité, et les longues heures que tu passes à relire mes documents en anglais. Que ce mémoire que je leur dédicace soit pour eux un signe de remerciement pour tout cela.

Abstract

Several software-centric development methodologies have been proposed during the last decade. They either focus on code first (Extreme programming, Agile development process), or on models: Model Driven Engineering is an initiative from the OMG to promote the use of graphical modeling languages at different abstraction levels. MDE targets the enhancement of main software engineering criteria, including maintenability, extensibility, reliability and development time. OMG assumes the definition of domain-specific languages based on profiling or meta-modeling techniques, and suggests to use the Unified Modeling Language for that purpose. Model transformations techniques ensure transitions between modeling abstraction levels, simulation, formal verification, and code generation.
One of our former contribution, TURTLE, was a pioneer of formally defined UML-based domain-specific graphical languages. TURTLE was defined at the end of the 90's for the design of time-constrained systems. Its main strength relies in its semantically enhanced relations between UML classes, in its timing operators at state machine levels and in its formal semantics defined in RT-LOTOS. Based on this initial contribution, this manuscript addresses the whole development cycle of complex embedded systems. In particular, the complexity of those systems encourages to take the appropriate design decisions as soon as possible in the development cycle, so as to avoid costly late system re-engineering. This remark advocates for an early use of formal description techniques in the development cycle, e.g., at software / hardware partitioning, and then, all along next stages (analysis, design, deployment) until executable code is generated. The poor acceptability of formal methods in industry also led us to propose techniques to ease formal proofs at each methodological stages. Finally, our contributions can be summarized in four main topics.
First, the definition of software-centric methodology covering all stages of safety-critical software development: dimensioning, analysis, design, deployment and code generation. TURTLE was extended to cover all these phases, with formal verification in mind.
Second, another UML profile was defined to specifically address the hardware-software partitioning stage. DIPLODOCUS is based on a set of abstractions, and follows the Y-chart approach. DIPLODOCUS comes with a very efficient simulation engine that takes into account hardware components on which functional entities have been mapped. A coverage-enhanced simulator tries to offer a good trade-off between model coverage and combinatory explosion.
Even if system designs are now commonly done with graphical models, properties to be proved are still captured with textual languages. Thus, our third main contribution proposes to model properties within UML-based views, and to automatically derive property models into either observers or properties formulae.
Raising security threats on safety-critical embedded systems advocates for taking both safety and security requirements into account as early as possible in system development. Thus, our fourth contribution proposes a new SysML environment - named AVATAR - specifically defined to take into account both safety and security properties in graphical models, and to automatically prove both kinds of properties from the same system models.
All environments mentioned above have been fully integrated and implemented in an open-source toolkit named TTool. Finally, this research work has received grants from industrial partners, and has also been publicly funded within national and European projects.

Contents

Chapter 1

Introduction

Most software today is very much like an Egyptian pyramid with millions of bricks piled on top of each other, with no structural integrity, but just done by brute force and thousands of slaves.

Alan Kay

1.1 Context

This document overviews research activities I've conducted over the last ten years, first during my Post Doctoral term at Concordia University, Montreal, Canada, and then as an Assistant Professor at Telecom ParisTech, Sophia-Antipolis, France. My research work mostly focused on techniques to efficiently model and verify complex embedded systems, with a particular emphasis on software hardware / partitioning of Systems-on-Chip, and on security issues in embedded systems. Indeed, during the last decade, techniques for designing embedded systems have been stimulated by the increasing complexity of embedded applications that are expected to answer to external stimuli within given timing constraints. Additionaly, the complexity of those systems also pushes to take the appropriate design decisions as soon as possible in the design cycle, so as to avoid costly late system re-engineering. This remark advocates for an early use of formal description techniques in the development cycle. Nonetheless, once first decisions - e.g., software / hardware partitioning - are proved as correct, a formal methodology should enforce those first decisions all along remaining methodological stages, until the implementation stage.
Unfortunately, industrial practitioners are rather reluctant to use formal description techniques. On the contrary, non formal modeling languages are widely accepted to document and exchange information during system development: UML [151], SysML [152], AADL [165] are a few examples of languages that are commonly used and supported with mature toolkits. This situation has obviously stimulated research work on coupling those languages with formal description techniques in order to enable the formal verification of diagrams described in precited languages (see, for example [2, 164, 183, 184, 58, 146]).

1.2 Problematic

A common way to add formality to UML is to define a "profile," i.e., a customization of the OMG-based notation, in order to meet specific needs in a particular application domain. We have first experimented with the definition of UML profile named TURTLE (Timed UML and RT-LOTOS Environment) [25, 19]. The profile has a formal semantics given in terms of RT-LOTOS. The major advantage of backing TURTLE on RT-LOTOS lies in the possibility to reuse RTL [57], a formal validation tool developed by LAAS-CNRS for handling real-time operators of RT-LOTOS. TURTLE has been defined to be able to model real-time critical systems, and perform formal verification over those models. In particular, it has been used in the scope of the dynamic reconfiguration of space-based embedded software [35]. The first version of TURTLE unfortunately faced the following limitations, which were progressively identified, and then addressed:

1. **Full methodology support**. TURTLE was limited to "design" diagrams. By design diagrams, we mean the definition of the software architecture of a system - in terms of classes-, and the behaviour of these classes (state machines). As we mentioned before, there is a clear need for adding formality to all methodological steps, including requirement capture, system analysis, and system deployment and prototyping. In particular, since requirement capture was omitted in the first definition of TURTLE, it was not possible to trace requirements when performing formal proofs from TURTLE diagrams, nor was it possible to enter given properties to be proven in a high-level language, thus forcing designers to rely on low-level property languages (e.g. CTL, PSL). This mix of high level languages (e.g., UML) and low level ones (CTL) obviously reduces the interest in semi formal approaches.

2. **Abstraction of computation durations**. TURTLE designs abstract functional and non functional delays with non-deterministic temporal operators. That is, computations' duration can easily be abstracted. Unfortunately, those durations may not be known a priori, leading to the need to profile those algorithms (and, so, implement them), or to make proofs for oversized non-deterministic time intervals.

3. **Hardware platforms and implementation issues**. TURTLE mostly addresses software architecture, casting away hardware matters and system partitioning issues. More generally, all underlying layers of software, including middleware, operating systems, processors, buses, memories are not taken into account when performing formal proofs from TURTLE models. System implementation and evaluation are also not handled. That is, TURTLE designs have to be translated by "hand" to the target language (e..g, C, Java), and then evaluated. Again, this approach reduces the interest to use formal verification techniques.

4. **Security concerns**. Security has become a major issue in many embedded systems: mobile terminals, automotive systems, aeronautic systems, etc. Unfortunately, TURTLE does not offer any way to model security mechanisms nor supports the proof of security properties. In other word, a designer wanting to conduct both safety and security proofs from the same model needs to make two different models, and then needs to use two different proof techniques.

5. **Toolkit support**. Profiles need to be supported with a toolkit that can handle diagram edition, and can offer formal verification at the push of a button. The toolkit

issue is of critical importance when coming to the industrial acceptance problematic. A toolkit is also meant to demonstrate the practical efficiency of an approach.

1.3 Contributions and Outline

This document summarizes all contributions that we have worked on during the last years, in order to overcome issues mentioned in the previous section. Basically, contributions target the definition of an integrated and graphical environment to cover several embedded system development phases. Figure 1.1 represents all contributions represented on the usual V software development cycle. The initial contribution (TURTLE) is depicted within a red rectangle (Design, safety). New contributions are now more detailed with regards to this Figure 1.1, and with regards to the V development cycle.

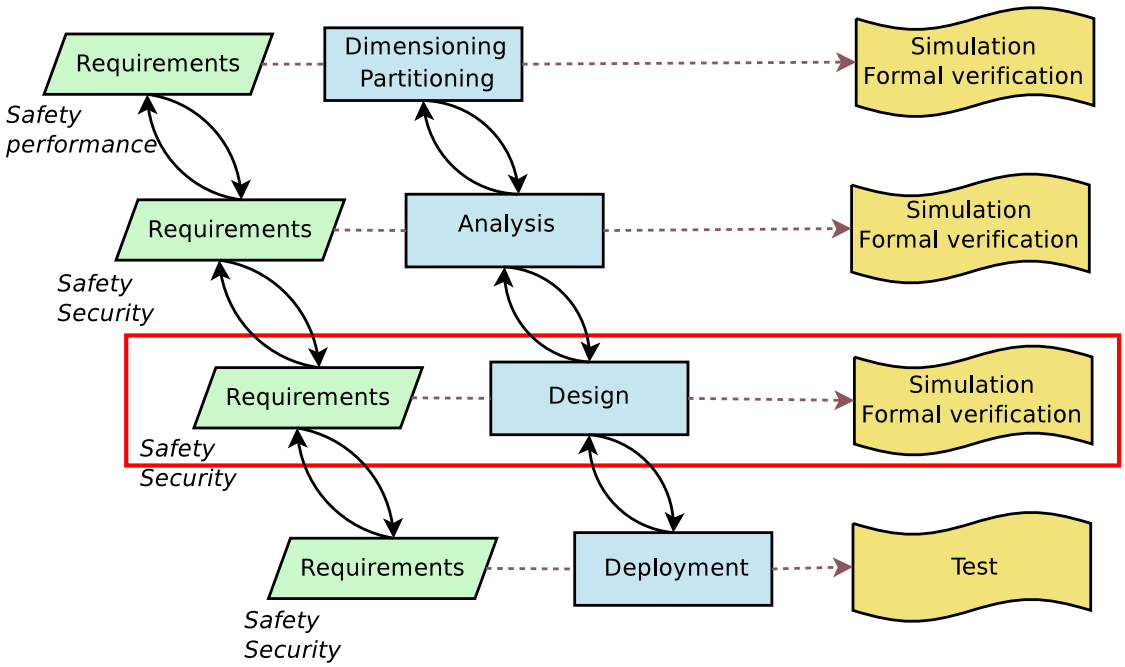


Figure 1.1: Contributions overview

Basically, those issues have been addressed as follows:

1. TURTLE was enhanced with other methodological phases (see Figure 1.2) so as to cover an extended part of embedded system development: system dimensioning, system analysis and system deployment stages. System dimensioning is based on UML deployment diagrams limited to the modeling of traffic sources, traffic destinations and network routers. System analysis extends the UML analysis diagrams with non deterministic temporal operators and preemption relations between scenarios. System deployment leverages UML with explicit and concrete scenarios and protocols between execution nodes (e.g., TCP/IP protocol, UDP protocol, or middleware-based communication). The three stages have been given a formal operational semantics. Moreover, we have also introduced automatic design synthesis from analysis models, and automated executable code generation from deployment diagrams. At last, we have proposed patterns covering different methodological stages so as to assist engineers in their modeling tasks. All those contributions are described in chapter 2.

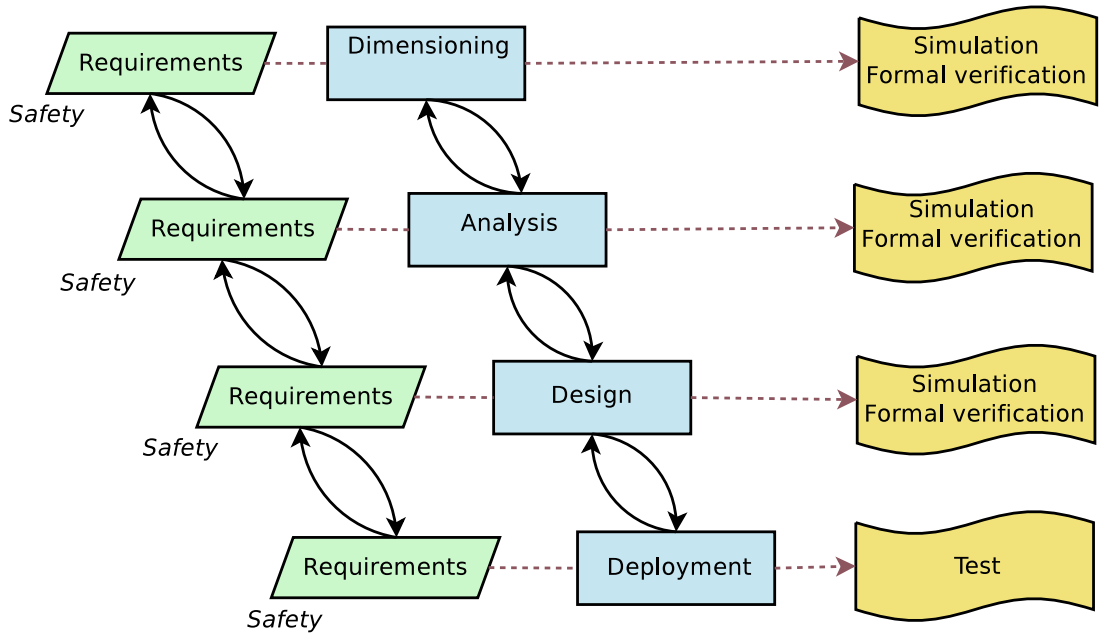


Figure 1.2: Enhancement of TURTLE

2. We decided to put a particular focus on the system partitioning phase. Indeed, the TURTLE dimensioning phase cannot be applied to complex embedded systems or chips in which the dimensioning of computations resources (e.g., CPUs and hardware accelerators) is at stake, of whenever the software / hardware architecture is not obvious or known in advance. Indeed, the Design Space Exploration we introduce is a process dedicated to the analysis of various functionally equivalent software/hardware implementation of systems specification. The result of this process shall be an optimal hardware / software architecture. The DIPLODOCUS profile was introduced to offer engineers with the necessary tools and diagrams to determine this optimal partitioning (see Figure 1.3). Basically, DIPLODOCUS is based on high level and abstract UML models, that can be simulated or validated at the push of a button. Models clearly separate functional issues from hardware architectural ones, making it easy to experiment with different partitioning schemes. Fast simulations and formal verification are the two techniques that can be used to investigate a given mapping of functions over hardware architectures. The overall DIPLODOCUS approach is explained in chapter 3.

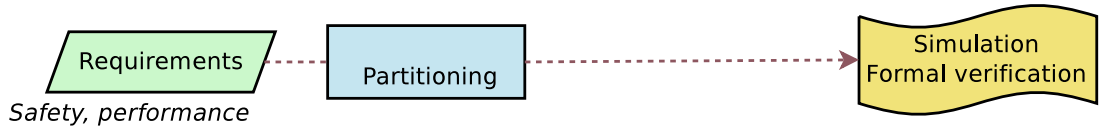


Figure 1.3: System partitioning: the DIPLODOCUS environment

3. Whether a specific methodological stage is used (e.g., the partitioning one), or a sequence of methodological stages (e.g., TURTLE analysis, design, deployment), the verification of methodological models is performed according to a set of properties. An important contribution is the possibility to directly express system requirements

and properties in the model, rather than with low level languages (see Figure **??**). To do so, we have experimented with different approaches. At first, time-related properties are expressed within UML Timing Diagrams, and automatically derived as observers in TURTLE design diagrams. Those temporal properties can be related to non formal requirements expressed in SysML Requirement Diagrams. This first contribution was followed with a more generic one aiming at expressing both logical and temporal properties. To do so, the SysML parametric diagram is used, and can relate design elements (e.g., attributes, signals) to build up complex logical and temporal properties. Then, properties can be automatically derived either to observers or into specifications in given well known verification languages (e.g., CTL like languages). Requirements and properties models and semantics are detailed in chapter 4.

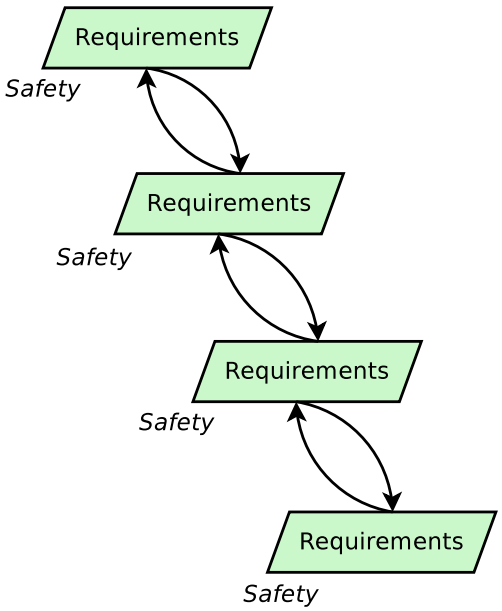


Figure 1.4: Requirements and property modeling

4. TURTLE is mostly focused on safety-related properties, and DIPLODOCUS targets both safety properties and performance related metrics. But, as previously mentioned, security is now an important aspect of many embedded systems, and shall be explicitly taken into account in all system development stages. This issue is addressed by a new SysML environment named AVATAR (see Figure 1.5). The latter borrows most TURTLE concepts in a SysML way, thus avoiding the mix of SysML (e.g., with requirements diagrams) and UML (e.g., class diagrams) in the same methodology. AVATAR supports analysis, design, and prototyping stages. Moreover, in AVATAR, security requirements and attacks can explicitly be captured. Design diagrams are enriched with explicit security mechanisms and features (e.g., cryptographic keys). Authenticity and confidentiality properties can also be expressed, and furthered proved with an automatic translation to the ProVerif format. AVATAR still supports the proof of safety properties. At last, the TURTLE deployment stage has been enhanced in AVATAR with a prototyping stage. Chapter 5 summarizes these contributions.

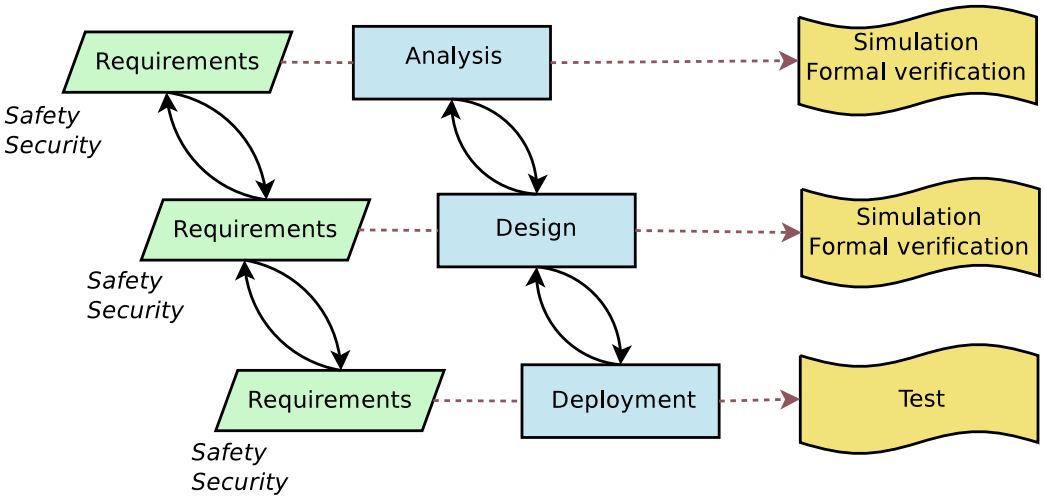


Figure 1.5: AVATAR

Chapter 2

TURTLE: An Environment for Embedded Systems Development

2.1 Context and outline

The increasing development of embedded applications which need to answer external stimuli under timing constraints has stimulated research work on real-time system design techniques. Also, life-critical systems and tremendous prototyping costs of complex real-time systems - e.g., aircrafts, satellites, and nuclear plants - has created potential conditions for deploying formal methods and *a priori* validation techniques that enable early detection of design errors in the life cycle of a system.

Despite the expected benefits of *a priori* validation based on formal modeling, industrial practitioners have been reluctant to use formal methods, even if a broad variety of modeling languages has been proposed to support the design trajectory of these systems. The lack of integration of formal techniques in development methodologies and the lack of skills on those techniques are probably the two main factors to explain that reluctance.

For the systems we target, we favor formal languages that explicitly take time into account and support verification of design solutions against temporal requirements, e.g., languages such as Time Petri Nets [8]. By contrast, an informal modeling language has received increasing acceptance in industry: the Unified Modeling Language [29], which is now an international standard at OMG (Object Management Group). The lack of formal semantics in UML has stimulated research work on coupling UML and formal modeling languages with the purpose of giving the OMG-based notation a formal semantics and to enable a priori validation of UML diagrams [15, 52, 51, 55, 56, 59, 69, 90, 76, 77, 78, 87, 92, 128, 170, 178]. To contribute to these solutions with a particular focus on timing constraints, we have introduced in [25] a UML profile called TURTLE (Timed UML and RT-LOTOS Environment), which has been successfully applied to the design of critical systems [35] [138].

However, the applicability of this first version of the TURTLE profile to the development of real-time and distributed systems is limited. Indeed, This first version was specifically targeting the design of software components in terms of classes: architecture, logical and temporal behavior. But, TURTLE could not explicitly be used to analyze a system, nor it could be used to design the low level architecture of a distributed systems - including its hardware parts -, and the communications constraints between various system components, e.g., Quality of Service parameters. To address these drawbacks, TURTLE

has been enhanced with several methodological stages: dimensioning, analysis, and deployment. The usage of TURTLE has also been grandly simplified with the definition of formally defined UML patterns.

Contributions to the TURTLE profile are presented as follows. The overall methodology is presented in section 2.2. TTool is the toolkit we have developed for supporting TURTLE: it is presented in section 2.3. Dimensioning, analysis, design and deployment stages are then presented in sections 2.4, 2.5, 2.6 and 2.7, respectively. Modeling patterns are then presented in section 2.8. Section 2.9 concludes this chapter.

2.2 Overall methodology

TURTLE now defines the following four-stage method:

1. **System dimensioning**. The main purpose of that stage is to parameterize the analysis and design diagrams with realistic upper bounds in temporal intervals. A "Dimensioning Diagram" describes the network in terms of traffic and equipments behavior, and a "Dimensioning-oriented Use Case Diagram" categorizes the flows conveyed by the network. The semantics of those diagrams is given throughout the *Network Calculus* Theory.

2. **System analysis**. Use-case driven analysis enables to identify the boundary of the system, the functions and services it offers, and the set of external actors it interacts with. Use-cases are documented by sequence diagrams. An Interaction Overview Diagram (IOD) relies on an activity diagram-like formalism to structure sequence diagrams. TURTLE sequence diagrams accept absolute dates and time intervals à la UML, and introduce timers à la UML/SDL. Two scenario instances communicate either asynchronously or synchronously.

3. **System design**. This stage was introduced during the first definition of TURTLE. Design diagrams include an object-oriented architecture depicted by a class/object diagram and a set of activity diagrams that describe the behaviors of objects. Associations between classes enable explicit modeling of task pairs that run in parallel or in sequence, rendezvous on gates or preempt each other. Support of time intervals allows TURTLE activity diagrams to include temporal indeterminism in the behavior of objects, so as to model e.g. non deterministic computation times or timers.

4. **System deployment**. Software classes defined at previous steps are grouped into components. Components may then be deployed over execution nodes using UML deployment diagrams. Executable Java code can be derived from that stage.

2.3 TTool

TTool fully supports the four-stage method described just before. Other UML toolkits could perhaps be tuned to support the TURTLE syntax. However, our experience has demonstrated that this tuning requires a strong involvement, and its is necessary to redo that tuning each time a new versions of the toolkit is released. In fact, TTool was first developed when no other alternatives were available (e.g., eclipse), and can easily be extended with profile modifications, or with new profiles. Also, as it is mainly developed in Java, it can be executed under most platforms.

Main differences between TTool and other UML front-ends interfaced with external verification tools are listed below.

- TTool offers user-friendly interfaces to formal verification tools and nicely manages the problem of linking verification results to the identifiers used in the TURTLE model. People with limited knowledge of formal methods may use TTool without reading a line of LOTOS [101], RT-LOTOS [57], or UPPAAL [42] code.

- TTool includes several code generators that enable application of complementary verification techniques, such as model-checking, transition system minimization and observers.

- All diagrams, but the use-case diagram, have a formal semantics. Therefore, formal verification applies not only to design diagrams but also to dimensioning, analysis and deployment ones.

- TTool enables to apply formal verification to analysis diagrams where other UML tools apply it to design diagrams exclusively. Errors may be detected without waiting for the design step to begin. Also, somebody unfamiliar with object-oriented design may restrict himself or herself to functional analysis. This was the case in several models, including the ones made in the scope of a project with UDCast.

- TTool bridges the gap between the analysis and design steps. It indeed includes a design diagram synthesizer which takes sequence diagrams as input and outputs objects and activity diagrams. Automatically generated diagrams may be extended manually and formally verified. This functionality is further discussed in section 2.6.3.

- At deployment step, TTool takes into account the distribution of components. Links between components are characterized by an asynchronous FIFO-based communication semantics, non-deterministic delays and a loss rate.

The rest of the section describes a verification approach that indifferently applies to analysis, design or deployment diagrams of the TURTLE profile. Those techniques also apply to other profiles, e.g. to the DIPLODOCUS UML profile.

2.3.1 Reachability analysis

TTool has been interfaced to verification tools that implement reachability analysis, a technique which computes the set of stable states the system may reach from its initial state. This subsection assumes a reachability graph may be computed in reasonable time.

CADP [85, 93], a tool developed for a version of LOTOS that does not include temporal operators. CADP enables quick generation of reachability graphs; nevertheless, temporal information of the original model is lost. Conversely, RTL [57] and UPPAAL [43] take the temporal operators - including non-deterministic temporal operators - of TURTLE models into account.

TTool not only invokes a verification tool it has catered with appropriate formal code, it also offers user-friendly interfaces to exploit the reachability graphs computed by CADP

or RTL. Thus, TTool computes statistics on states and transitions. It also identifies deadlocks as well as shortest and longest paths in the graph. Also, TTool uses *dotty* to display graphs. The latter contain identifiers that may not match with the identifiers used in the TURTLE model. Therefore, TURTLE provides a conversion table which allows to trace identifiers from TURTLE models to formal code and reachability graphs.

In practice, displaying a reachability graph is not sufficient to decide whether some property is met or not. The reachability graph of real-size systems may indeed have millions of states and transitions. Logic-based model checking and minimization are two complementary techniques offered by TTool and its companion tools.

2.3.2 Model checking

TTool offers a user-friendly interface to check for logic formulae (e.g. with UPPAAL). For example, to decide whether some UML action is reachable or not, or to study the liveness of that action, it suffices to right click on the corresponding action's symbol: The UPPAAL's verifier is invoked with corresponding CTL formulae, and the result of reachability / liveness properties is displayed. Temporal logic formulae in CTL may also be entered directly in TTool.

2.3.3 Minimization of labeled reachability graphs

A reachability graph may be transformed into a Labeled Transition System, a structure for which CADP implements minimization techniques based on trace or observational equivalences just to mention a few [57]. Graph's transitions associated with synchronization actions are labeled by action's name. Other transitions are labeled by "nil". The minimization process discards as much "nil" transitions as allowed by the equivalence relation and outputs a quotient automaton which gives an abstract view of the system's behavior. Minimization particularly applies to communication architecture validation. Given a protocol layer modeled in TURTLE, a labeled reachability graph is generated (RTL, CADP) and minimized by considering service primitives exchanges as observable events. The minimization thus outputs a quotient automaton of the service rendered by the protocol layer.

2.4 System dimensioning

2.4.1 Motivation

The modeling of distributed systems commonly relies on a three-layer architectural pattern where two or several protocol entities rely on a pre-existing communication service to provide their end user with a value-added service. Unfortunately, the pre-existing service is hard to characterize in terms of transmission delay and error rate. A survey of the literature indicates that authors commonly use empirical values. This generally leads to a space explosion problem inherent to reachability analysis techniques implemented by the formal verification tools linked to TTool.

The dimensioning stage proposes to bypass the problem using analytical realistic upper bounds, obtained with the Network Calculus formalism [130], for the pre-existing service. The objective is to address the dimensioning problem as early as possible in the development cycle and to reduce design and prototyping costs.

This work was lead in collaboration with ISAE and published in [31, 26, 32].

2.4.2 Related work

The increasing complexity of distributed systems and the need for architecture validation have stimulated research work on protocol modeling using Petri nets [70], formal description techniques (Estelle, SDL and LOTOS) and UML profiles (such as TURTLE) that bridge the gap between the UML and formal methods world. Many contributions in this area address language definitions, formal verification tools and code generators. A survey of the literature indicates that too little attention has been paid on methodological issues with some exceptions, such as [27].

Methodologies which apply to distributed systems commonly identify communication architecture validation as a fundamental issue, and reuses the three-layer pattern described in figure 2.1. Two or several protocol entities rely on some pre-existing communication service to offer to their upper users a value-added service. The pattern has extensively been used since the mid-1980's, including for protocol engineering based on UML [160].

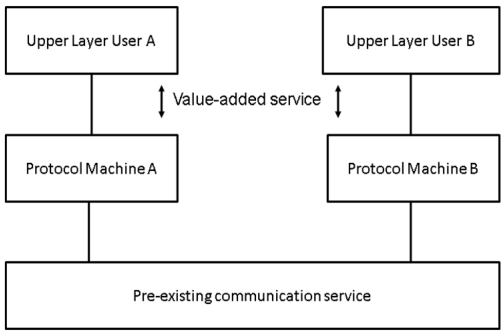


Figure 2.1: Three-layer architectural pattern

The increasing use of the UML gives protocol engineers a notation to share. The acceptance of UML among practitioners also depends on the capacity of UML tools to offer model analysis techniques. For instance, the proSPEX method [68] implements performance evaluation into TAU G2, a tool which supports a UML/SDL profile. De Wetre and Kritzinger promote a non analytical technique based upon simulations and application of queuing theory. Consequently the network parameters are approximated by lower bounds.

Unlike this approach, the one we have selected in the scope of TURTLE is based on the use of analytical realistic upper bounds to characterize the communication service (e.g. transmission delays, throughput of memory) thanks to the *Network Calculus* formalism. This theory is well adapted to controlled traffic sources and provides easily analytical upper bounds of the system's features. Our work joins the use of UML profile and the Network Calculus theory for dimensioning purposes. To our best knowledge, this kind of approach has not been addressed yet for distributed systems. However, it is worth to note that in terms of joint use of Network Calculus and a high-level modeling language, there are some existing tools like CYNC [167] which associates the Network Calculus with Matlab/Simulink. Moreover, the idea to compute bounds to come up with worst-case scenarios is not exclusive to the Network Calculus. For example, the synchronous language SCADE is proposed for that purpose in [129], but this contribution does not target distributed systems.

2.4.3 Network Calculus Fundamentals

The Network Calculus formalism [130] relies on *min-plus* algebra for designing and analyzing deterministic queuing systems where the compliance to some *regularity constraints* suffices to model the traffic. These constraints limit traffic burstiness in the network and are described by the so called *arrival curve* $\alpha(t)$, while the availability of the crossed node is described by a *service curve* $\beta(t)$. The knowledge of the arrival and service curves enables the computation of the delay bound that represents the worst case response time of a message, and the backlog bound that is the maximum queue length of the flow. The backlog bound helps configuring the memory of the system.

As shown in Figure 2.2, the delay bound D is the maximal horizontal distance between $\alpha(t)$ and $\beta(t)$. The backlog bound B is the maximal vertical distance between them. Two conditions make the bounds easier to calculate: (i) a linear arrival curve $\alpha(t) = b + rt$ with b the maximal burst and r the rate (we say that the flow is (b, r)-constrained); (ii) a rate latency service curve $\beta(t) = \max(0, R(t - T))$ with latency T and rate R. The obtained bounds are $\frac{b}{R} + T$ for the delay and $b + rT$ for the backlog.

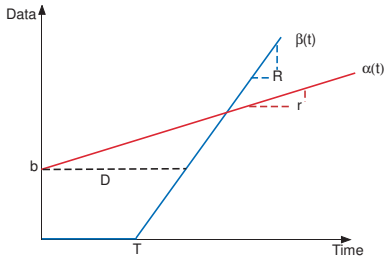


Figure 2.2: How to compute the maximum bound on delays and queues

This formalism gives an upper bound for the output flow $\alpha^*(t)$, initially constrained by $\alpha(t)$ and crossing a system with a service curve $\beta(t)$,using min plus deconvolution $\oslash$ where:

$$\alpha^*(t) = sup_{s \geq 0}(\alpha(t + s) - \beta(s)) = (\alpha \oslash \beta)(t)$$

The output arrival curve of the flow $\alpha(t)$ in the case of a linear input arrival curve $\alpha(t)$ and a rate-latency service curve $\beta(t)$ is simply $\alpha^*(t) = b + r(t + T)$.

2.4.4 General approach

A Dimensioning Diagram (DD) is created to describe the network in terms of traffic and equipments behavior. A dimensioning oriented use-case diagram, or DUCD for short, is introduced to document the DD and categorizes the flows conveyed by the network. The information contained in the DD and the DUCS is provided as input to the Network Calculus Software (NCS) that has been interfaced with TTool. NCS analytically computes upper bounds for network parameters such as communication delays and queues lengths (i.e. memory usage). The NCS output values are used at the design level to have an accurate model of the pre-existing service.

In UML, a deployment diagram is made up of execution nodes that may be stereotyped (e.g. by $<< PC >>$ or $<< Switch >>$) and pair-wise connected by links. A deployment diagram may also contain *artifacts* that identify software elements issued from a development process.

In TURTLE, DDs customize the concept of nodes, link and artifacts to model the equipments, traffics, routers (including routing tables of those routers), router-to-router interconnections, and equipment-to router-interconnections:

- *Equipments* are modeled with UML nodes stereotyped as $<< Equipment >>$.
- *Traffics* are modeled as $<< Traffic >>$ artifacts to be added to the nodes stereotyped by $<< Equipment >>$. Each traffic is characterized by a *type* (periodic, aperiodic), a *deadline*, a **minimal** and *maximal packet size*, and the *priority level* inside the router (four priority levels are supported). Figure 2.3 shows an example of using TTool for entering traffic information.
- *Routers* and *switches* are modeled with UML nodes stereotyped $<< Switch >>$, see e.g. switch0 on Figure 2.4. Each switch is characterized by a *scheduling policy* (Static Priority, First Come First Served), a *capacity* expressed in Mbs or Gbs, and a set of *entry / exit interfaces*. That capacity is taken into account if and only if the capacity of the links connected to the router is not set.
- *Routing information* is modeled inside a router, using $<< Routing >>$ artifacts to be mapped onto $<< Switch >>$ nodes. See, e.g., the artifact R0 of node switch1, Figure 2.4. A route is defined by a 3-uple (*entry interface*, `traffic`, *exit interface*). A routing artifact may contain one or several routes.
- *Links* between nodes model exit and entry interfaces of equipments and switches. At equipment level, multiplicity parameters may be associated to the interfaces, so as to model large numbers of similar equipments in a system (e.g. thousands of mobile phones).

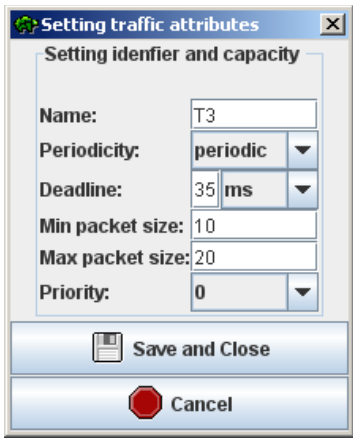


Figure 2.3: Entering traffic characteristics

2.4.4.1 Properties

The network calculus theory applies from DDs if the latter satisfy the following properties:

- The traffic routing must not contain any cycle.

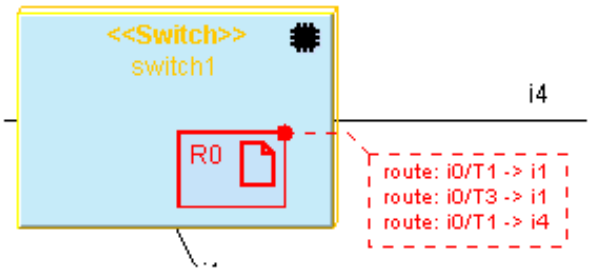


Figure 2.4: Characteristics of a *Route* artefact

- A multiplicity may be attached to a link between an equipment and a switch; it exclusively applies to that equipment. It is not permitted to attach a multiplicity to another type of link.
- For all traffics emitted to the network, there must be at least one *well-formed* path making it possible for the traffic to be sent to a destination equipment. Here, *well-formed* means that for all traffic entering a switch, (i) there must exist at least one output interface which differs from the input interface, and (ii) the routing information associated with the switches must allow each traffic to reach at least one equipment.

2.4.5 Results

The dimensioning phase of the TURTLE profile has been applied to several systems [31, 26, 32] for which upper bounds were succesfully calculated. Several contributions have addressed the UML graphical capture of system characteristics. Nonetheless, the translation to network calculus specification and the analysis of these specifications are still on-going works. New results are thus expected in the incoming year.

2.5 System analysis

2.5.1 Basics

Basically, a TURTLE analysis is as follows. Use-case driven analysis enables to identify the boundary of the system, the functions and services it offers, and the set of external actors it interacts with. Use-cases are documented by sequence diagrams. An Interaction Overview Diagram (IOD) relies on a activity diagram-like formalism to structure sequence diagrams. TURTLE sequence diagrams accept absolute dates and time intervals à la UML, and introduce timers *à la* UML/SDL. Two scenario instances communicate either asynchronously or synchronously.
Research work around the TURTLE analysis stage have been published in [23, 24, 27, 36, 20, 21, 22], and used in the scope of several projects, including the modeling and verification of multimedia broadcasting (UDCast).

2.5.2 Semantics

A TURTLE analysis does not change much with a "regular" UML analysis. The main strength of a TURTLE analysis relies in the formal semantics that is given to TURTLE

analysis diagrams. On the one hand, use case diagrams are provided for documentation purpose only, on the other hand Interaction Overview Diagrams and scenarios are given a formal semantics in RT-LOTOS. A set of interaction Overview Diagrams and scenarios they referenced can indeed be translated into a TURTLE design - see section 2.6, and then, the semantics of TURTLE design can be reused. Design synthesis is further explained in section 2.6.3.

2.5.3 Excerpt of a specific contribution

Let us present the interest of one specific TURTLE analysis operator: the preemption operator. This operator can be used at Interaction Overview Diagram level to model that a given scenario may preempt another one as soon as one action in the preempt scenario can be performed (see Figure 2.5). Whenever an preemption occurs, the preempted branch of scenario is totally interrupted forever, and the branch with the preempt scenario is executed. This operator is particularly useful to model situations where a protocol disconnection can occur at whatever moment after a connection has been setup.

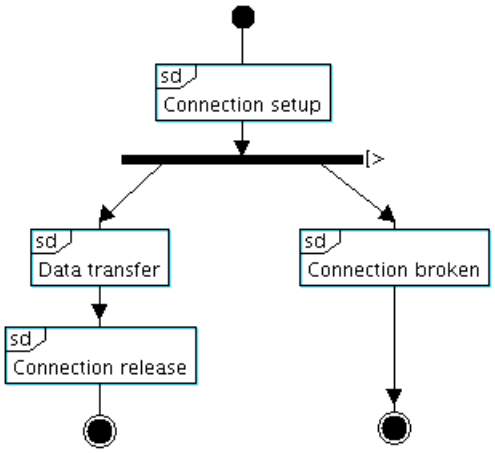


Figure 2.5: Use of the preempt operator in an Interaction Overview Diagram

2.6 System design

2.6.1 Basics

TURTLE designs have been first introduced in, and further explained in two Ph.D. thesis [35] [138], and in several papers, either to present the core of the design diagrams [25, 65, 28, 66, 19], or to present extended versions [137, 64, 63], or toolkit issues [21], or the application of TURTLE to the modeling of complex systems [29, 21].

Basically, A TURTLE class diagram is made up of "normal" classes and stereotyped classes that we call *Tclasses* (TURTLE Class). Communications through public attributes or method calls are limited to communications between a Tclass and a normal class, or between two normal classes. Communication between two Tclasses uses so-called "gates." A gate is a particular Tclass attribute of type Gate. A gate can be used for synchronized communication with another Tclass, or for an action internal to a Tclass.

The internal behavior of each Tclass must be described using an activity diagram. The TURTLE profile extends UML activity diagrams with synchronization operators and

temporal operators. The former are used to express synchronizations which are internal to Tclasses, or synchronizations between Tclasses. The latter are used to describe temporal constraints that apply exclusively to the internal activities of Tclasses. In its activity diagram, a Tclass T may perform a call on a gate g. Three cases may apply:

- g is not synchronized with any other gate. In that case, a call on this gate models an internal action
- g is internally synchronized. The synchronization occurs when two subactivities of a Tclass are ready to execute action "g."
- g is externally synchronized (g must be declared as public). In that case, T must be linked with a Synchro composition operator to another Tclass, and g must be specified as synchronization gate in an OCL formula attached to the Synchro operator. Any relation attributed by a Synchro operator must be decorated by an OCL formula stating which gates are connected together.

Four temporal operators are introduced at activity diagram level: a deterministic delay, a non deterministic delay, a time-limited offer on a gate, and a time capture operator.

2.6.2 A few extensions

All details on TURTLE design operators and semantics can be found in [35]: we do not wish to put too much emphasis on this definition in this document since most of it was realised during my Ph.D. work. However, there were a few extensions defined and published afterwards to enhance TURTLE designs [137]. Those extensions represent in fact modeling schemes which were frequently used to capture real-time system behaviours: requesting a task to execute, specifying a task period, modeling task suspension and reactivation.

- **Invocation**. A task can request another task to execute once. This semantics will be later reused in the definition of the DIPLODOCUS profile (see chapter 3).
- **Periodic tasks**. A task can be specified as being periodic. The deadline of the task can be specified as well, and further verified at validation step.
- **Task suspension and reactivation**. Semantically speaking, this operator is the most complex one. Indeed, when a task is suspended, its temporal operators continues to elapse, that is, the system must continue to execute. The underlying RT-LOTOS semantics is based on the "at" operator.

Two other modeling facilities have been introduced later:

- **TData**. A TData represents a data structure built upon default TURTLE types. This facilitates complex data exchanges (e.g., a message on a communication medium).
- **TObject**. A TObject represents an instance of a TClass, with a customization of attributes value.

TURTLE designs were implemented in TTool beginning of 2004, and were used in several academic and industrial case studies, in particular with Thales Alenia Space in the scope of the modeling of space-based embedded systems dynamic reconfigurations [35]. TURTLE designs are also at the origin of all other TURTLE stages and all other semi formal modeling approaches we defined afterwards, including DIPLODOCUS [33] and AVATAR [112].

2.6.3 Design synthesis

2.6.3.1 Problematics

Diagram coherency is a topic of utmost importance to detect modeling issues. Our synthesis approach proposes to generate design diagrams from analysis ones, i.e., from interaction overview diagrams and sequence diagrams. This work is settled on contributions made to generate SDL diagrams from HMSCs [4]. Well known techniques for design synthesis are discussed in [13]. Some approaches do not check for the validity of scenarios before giving them as input to the design synthesis engine. The idea is thus to rapidly obtain a first design model that a designer can further rework, and validate. Other proposals validate these scenarios before the synthesis stage, so as to ensure system behavior conformance between the design and scenarios. Indeed, a set of scenarios may be inconsistent, may contain blocking points (e.g., deadlocks), or more generally, the scenarios may be not implemented with the synthesis approach. This problem is called *realisability* [11], or *implementability* [111].

2.6.3.2 TURTLE design synthesis

Basically, our synthesis technique is as follows [24]: it takes as input one main Interaction Overview diagrams (IOD) that may reference scenarios described with Sequence Diagrams (SDs). The synthesis process outputs one TURTLE design, i.e., a TURTLE class diagram (CD) and one activity diagram for each Tclass of CD, i.e., a set of ADs. Synthesis algorithms are quite complex because UML analysis and design diagrams have strongly different operators. Their implementation represent around 6000 thousand lines of Java code in TTool. The general philosophy of these algorithms is as follows:

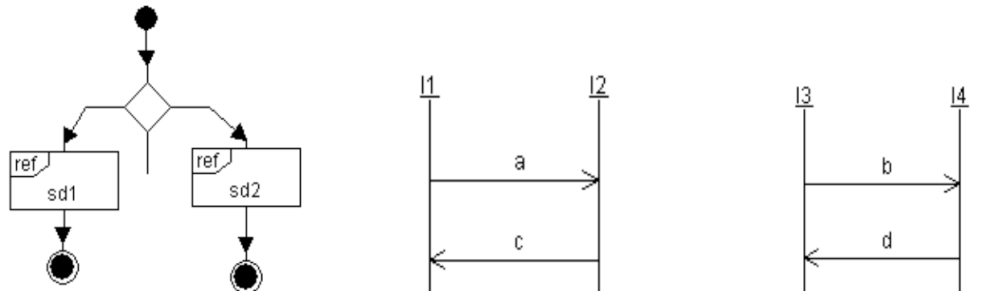
1. For each instance I_i modeled in at least one SD, a Tclass T_i is generated in CD.

2. Synchronous messages can easily be translated using TURTLE "Synchro" composition operators.

3. For each asynchronous message m_k sent by instance I_i to instance I_j, a communication channel is created between T_i and T_j. This channel is modeled with two intermediate tclasses Tin_ij_mk and $Tout\_ij\_mk$. These classes are synchronized with T_i and T_j, respectively. Tin_ij_mk and $Tout\_ij\_mk$ synchronizes together. Channels are modeled as totally ordered with a buffer at destination side. No delay are applied on messages.

4. Each timer of sequence diagrams are modeled with a specific class offering three synchronization gates: *set*, *reset* and *exp* (expiration), as defined in the MARTE UML profile [154]. Each class using a timer is synchronized with the class corresponding to the given timer. A similar approach is used to model time constraints.

5. The behavior of T_i is built as follows. For each event $evti_j$ of instance I_i a sub activity diagram is built. These sub activity diagrams are connected together as follows. If two events $evti_j$ and $evti_k$ are in sequence in the sequence diagram, then, there are connected together. Otherwise, the two subactivities are not ordered, (co-regions, different scenarios, etc.) are connected taking into account operators of Interaction Overview Diagrams: choice, preempt, etc.



(a) Interaction Overview Diagram (b) Scenario sd1 (c) Scenario sd2

Figure 2.6: Non implementability example

2.6.3.3 Implementability of TURTLE analysis

Given a TURTLE analysis, i.e., a set of TURTLE scenarios interconnected with Interaction Overview Diagrams, an important issue to solve is whether the generated design has exactly the same behaviour as the one expressed by the corresponding analysis diagrams. Indeed, a set of scenarios describes a global and centralized view of the considered system, i.e., all instances are expected to execute the same scenario at the same time. On the contrary, a design is composed of a set of independent units communicating with channels. In some modeling schemes, our synthesis approach may unfortunately generate a TURTLE design that contains more execution traces than the analysis taken as input by the synthesizer. Let's consider an example. Two scenarios can be exclusively executed: either scenario *sd1* or *sd2* (see Figure 2.6a). In scenario sd1 (Figure 2.6b), I1 sends message a to I2, and then, I2 responds with message c. In sd2 (Figure 2.6c), I3 sends message b to I4, and then I4 responds with message d. The system assumes that if I1 sends message a, then I3 mustn't send message b. On the contrary, if I3 sends message b, I1 must not send message a. But our translation algorithm cannot detect such scheme, and translates I1 and I3 without this dependency, i.e., the distributed choice modeled in the IOD is not translated. Other non implementable schemes are provided in [24]. Automatically identifying some non implementable situations is still an open issue, even if it is has been demonstrated as non decidable in the general case [142] [12]. Contributions published in [192] and [62] are a starting point.

2.7 System deployment

2.7.1 Context

The deployment phase consists in mapping "software components" on execution nodes. In TURTLE, those software components are usually built upon classes extracted from the design.

High-level designs such as TURTLE models have to be refined into more concrete designs before reaching the implementation phase. One of the major steps in this process is the identification of the components and their deployment. We are interested in defining components, deploying them, and studying the properties of such deployments as early as possible in the development life cycle. Since UML deployment diagrams have no formal semantics, a formal investigation of properties of potential deployments is impossible. This

section explains how we have formally defined TURTLE components and how we support their deployment over execution nodes.

UML deployment diagrams depict the "actual physical" configuration of a distributed system. Thus, a UML deployment diagram can be seen as a low-level design of a system, i.e. a design closer to the real system implementation. Software methodologies scarcely use deployment diagrams, and when they do, their role is limited to documentation.

UML deployment diagrams suffer other drawbacks. First, communication links, if stereotyped, remain imprecise because no clue can be given whether, when several links are modeled between two nodes $n1$ and $n2$, which link is used for sending a message from a component on $n1$ to a component on $n2$. Second, tough UML deployment diagrams have been introduced with distributed systems in mind, they do not offer any features for modeling large distributed systems, i.e. systems with a high number of nodes. At last, UML deployment diagrams are used for documentation purpose. We do think this is an important drawback since the deployment of components among nodes may introduce new errors inherent to distribution characteristics. If UML 2.0 has introduced a composite structure diagram to address some of the drawbacks listed above, this diagram does not support communication links with various characteristics (FIFO, delay, jitter, and so on). Also, composite structure diagrams do not include any operator or concepts to explicitly model physical nodes.

UML deployment diagram operators are:

- **Nodes**. These nodes represent the various physical locations of the system under consideration.
- **Software components**. According to the UML 1.5 standard, software components are deployable components that encapsulate functions and that offer interfaces to their environment. An instance of a software component can execute at a time on only one node.
- **Communication links**. A communication link connects a node to another node, and is often stereotyped to indicate its type (e.g. «Ethernet»);
- **Dependency links**. A dependency link connects two software components.

2.7.2 Basics of TURTLE deployment

Deployment diagrams are obviously suitable for the description of concrete distributed architectures: we have given them a formal semantics with the purpose to perform formal validation of low-level UML designs. More precisely, we have proposed the following enhancements for UML deployment diagrams [36]:

- Software components are built upon UML classes, TURTLE classes and TURTLE composition operators. They can be formally validated;
- A multiplicity can be used at node level. This makes it possible to use a single graphical node to describe several - and even many - physical nodes on which different instances of the same software components run;
- Communication links connect component interfaces and not nodes. Also they can be characterized with Quality of Service parameters, such as transmission delay and jitter.

UML has no formal semantics, and therefore any diagram element might be considered as given for documentation purpose. This remark particularly applies to links in deployment diagrams. Since our objective is to take communication constraints between components into account during formal validation, a formal semantics has also been given to links between components. Links are supposed to be unidirectional and asynchronous. A link is graphically represented by an association - with navigability - between the two interconnected interfaces. We attribute these associations with four additional and optional parameters given in an OCL formula:

1. A minimal transmission delay (*min_ delay*)
2. A maximal transmission delay (*max_ delay*)
3. A bandwidth indicated by the maximal number of messages carried at the same time on the link *(max_ msg*)
4. An average loss rate given as a percentage (*average_ loss_ rate*)

2.7.3 Proving properties

It is the designer's task to build the software components of the deployment diagram, and validate properties of the model using observers. Moreover, a low-level design contains not only software components, but also links that also have to be observed. While the observation of a software component can be done in the same way as a Tclass, i.e. with non-intrusive observers included in software components, observing a link is not that easy as discussed hereafter.

In TURTLE designs, observers are modeled at class diagram level [80]. In deployment diagrams, we introduce probes to observe properties related to links: a probe can be attached to the observed link on the deployment diagram. A probe is defined as a stereotyped Tclass (called Probe). Each particular probe of the system must be defined as a Tclass, which inherits from the Probe Tclass (a probe may be identified with the «probe stereotype»). All Tclasses inheriting from Probe must be declared and modeled at class diagram level. If declared, they can be used at deployment diagram as follows. Let us assume that P is a «Probe» Tclass. P can be used as a probe for as many links as desired at deployment diagram level: the class must simply be used as an associative class for the considered link (an example is provided later on in the section). Note that only one probe can be attached to a link.

Several gates are declared protected in the Probe Tclass: *send*, *receive*, and *lost*. They can be used as desired by probes to observe a link. Gates used for observation must be listed in an OCL formula along with the observed link.

2.7.4 Code generation

Code generation is particularly interesting over a deployed model since this model is expected to be more concrete than a non deployed one. Code generation is mostly intended for prototyping purpose. It can also be used for generating a whole system implementation since behaviours not meant to be put in the model can be added as raw code in TURTLE models. Raw code is obvisouly not considered for formal verification purpose. The process we have defined - and more thoroughly explained in [27] - takes as input a TURTLE deployment and generates a set of Java classes.

A deployment is made upon a set of execution nodes on which are mapped software components. The code generation process we have defined can generate code for software components (recall: a software component is a gathering of TURTLE classes), and can also generate code for the interconnection between these components.

2.7.5 Code generation for components

Generating Java code from a TURTLE component requires to translate the structure of TURTLE classes into a set of Java tasks (i.e., Java threads), and to translate the behaviour of those classes into a behaviour of Java tasks. On the one hand, most TURTLE operators have a quite easy to implement counterpart in Java. For example, a sequence between two TURTLE classes can be translated as a synchronization between the end of a Java task and the beginning of another one. Similarly, loops, regular choices, variable modifications have a quite direct translation in Java. On the other hand, some operators don't have a direct translation. For example, non deterministic intervals, synchronization on gates, and non deterministic choices. Indeed, those operators are commonly used to abstract a behavior that the person in charge of the model do not wish to describe more concretely. For example, a non deterministic interval may represent the temporal bounds of a given algorithm computation. It could be translated as a random waiting in a Java task. A more interesting translation is obviously to make a call to an external code containing an implementation of that algorithm. For synchronization purpose, we have implemented our own Java library, relying on Java synchronized objects and method calls.

2.7.6 Code generation for links between components

We have given the possibility to enrich links between nodes with the specification of a given communication protocol, e.g., TCP/IP, UDP, or even more complex protocols implemented in middleware (e.g., CORBA, RMI). Whenever a TURTLE component makes a call on a gate connected to a component located on another execution code, the following procedure is used:

- The protocol is initialized, if not already. For example, a TCP socket is opened, or a CORBA stack is instantiated.

- The data exchange is performed, using a message sending to the destination site, or calling a remote method (RMI). It can be executed only in one-way approach, contrary to synchronizations between TURTLE classes.

We have settled libraries for performing those two steps with an easy-to-use API, thus facilitating the integration of other protocols.

2.7.7 Results

Formal verification and code generation has been succesfully experimented in several systems, including on models issued from European projects in which many execution nodes (e.g., satellite telecommunication system, see [27]). The deployment of software components over execution nodes is also at the root of DIPLODOCUS [33].

2.8 Patterns

2.8.1 Context

Since the TURTLE profile has been defined a few years ago, it may now be considered as stable in its syntax and semantics definition. But the way TURTLE can be efficiently used is still an open issue. More generally, patterns have been proposed to enhance the usability of modeling languages. Since the first reference book on design patterns [84], the two contributions [74] and [162] have shown their use for real-time systems and communication applications, respectively. However, the sharing of common artifacts among practionners is a first step that was further enhanced with the formalization of patterns. Thus, [124] combine patterns and temporal logics. [131] demonstrate how to efficiently use B patterns in a correct-by-design modeling approach.

A second issue is the efficient use of those patterns in a toolkit. Indeed, these patterns are meant to play the role of a *modeling assistant*. For example, *SDL Pattern Approach* [86] has been implements in the SPT toolkit, itself integrated as a module of TAU G2 [99]. SDL [102] is a language targeting the modeling of protocol architecture, protocol control and protocol data. SDL/MSCs and TURTLE share similar capabilities for analysis and design phases. SPT is specifically dedicated to the design of protocols. The user can easily modify basic protocol features - and modify identifiers used in those features - so as to build complex protocols. On its side, TURTLE patterns are not dedicated to one given software engineering domain: they obviously targets protocols, but also real-time systems, multimedia applications, and so on. Also, SPT forces to rely on non blocking asynchronous communications, whereas TURTLE synchronous scheme can easily be used to create complex asynchronous communication schemes.

2.8.2 Contribution

Our approach focuses on formally defined patterns for the analysis phase [20, 22]. Each pattern is built upon two diagrams that serve as a frame to use the patterns. Thus, a pattern contains one Use Case Diagram and one Interaction Overview Diagram.

We have then defined rules on how the two provided diagrams can be used and extended to make a custom system. For example, in a UCD, it is possible to add sub functions to use cases, and then inherited or optional functions can be added to those sub functions. There are also subparts of those diagrams that must be filled up. For example, IODs contains references to other empty IODs that must be completed, as explained in each pattern. Basically, a pattern is expected to be used as follows:

1. A user selects a pattern P according to the system he wishes to implement. For protocols, we have defined three patterns: connected mode, unconnected mode, group communication [22]. Some patterns are integrated to TTool, additional patterns can easily be built in TTool.

2. Patterns can be modified according to a set of rules defined in [22]

3. Once modified, patterns are first filtered so as to remove parts that have not been used (filtering), and parametrized parts of patterns are instanciated.

4. A syntax analysis phase checks whether the so-obtained analysis follows the TURTLE analysis metamodel.

5. Our patterns offer properties satisfied by design. Other properties might be evaluated using the RTL, CADP or UPPAAL code generator of TTool only if the diagrams are realizable [24].

6. The last step is the automatic design synthesis i.e. the generation of a design equivalent to the analysis, that is, a first design is directly built from the used patterns.

2.8.3 Results and discussion

Patterns have been used to efficiently model group communication as defined in the *Maestro* project (several communication medium, routing between beams, group managements, etc.). Most parts of the system analysis stage were performed in less than a day using patterns. Formal verification was limited to a reduced number of group members. However, several problems were encountered using patterns.

- We cannot guarantee the implementability of systems built from patterns [24]. Thus, when modeling systems from those patterns, we have to avoid using distributed choices [24]. This strong constraint forces us to rely on synchronous messages rather then on asynchronous ones.

- Provided patterns assume that whenever a network connection is cancelled, another one can occur, i.e., there is a loop between the end of a session, and a new session. This stresses the formal verification stage that may have to handle too many execution states. In the case of the Maestro project, the use of patterns with session loop induces more than 3 million states in the reachability graph, and only 30k states in the version without loop. A solution to this problem would be to offer more correct-by-design properties thus avoiding the formal verification stage (e.g., liveness properties).

- Pattern parametrization is limited to given modeling schemes. More modeling schemes could be supported if the TURTLE profile was relying on a more component-oriented modeling scheme. Work has been conducted in that way in [8] and implemented in TTool.

2.9 Conclusion and perspectives

TURTLE supports most system modeling stages, from system dimensioning to system deployment and code generation. Formal verification can be applied from all TURTLE diagrams, including analysis and deployment ones. TTool offers a friendly Graphical User Interface to edit TURTLE diagrams, and perform formal verification at the push of a button. TURTLE has been successfully used for the modeling and proof of systems, including space-based embedded systems [35], automotive systems [83] [82], aeronautics systems [21], telecommunication systems [176] [67], and UML training in several worldwide academic institutions.

As explained in the introduction, TURTLE has been enhanced to explicitly support hardware constraints, to offer a nice way to integrate safety requirement and properties, and to also more explicitly support security properties. Next chapters focus on those three aspects.

Chapter 3

Design Space Exploration: the DIPLODOCUS Approach

3.1 Context

Recent Systems-on-Chips are expected to deal with complex applications at low cost (silicon area, power consumption). Design Space Exploration is a process to analyze various functionally equivalent implementation of systems specification. The result of this process shall be an optimal hardware / software architecture with regards to criteria at stake for that particular system (e.g., cost, area, power, performance, flexibility, reliability, etc.). The traditional top down methodology starts with an informal description of a system from which a reference model is first developed. Typical languages for these first models are Matlab, C or C++. That first model is verified against functional correctness, and might as well be used to drive first performance evaluations. This step is then usually followed with the proposal of several system software and hardware architectures. The most suitable one is selected according to criteria we've already mentioned before.
This design space exploration step is of utmost importance. Indeed, if critical high-level design choices are invalidated afterwards because of late discovery of issues (performance, power, etc.), then it may induce prohibitive re-engineering costs and late market availability. Also, the cost and the time to perform this exploration process shall be must lower than developing the real system, while being accurate enough to take the right decisions. And so, the cost to make the necessary system models, and the cost to evaluate those models shall be low.
To address these issues, we have introduced a few years ago the DIPLODOCUS UML profile [188] [33]. In DIPLODOCUS, the designer is supposed to model in totally separated views the application and the architecture of the targeted system. Thereafter, a mapping stage associates application and architectural components. To assess the satisfiability of both functional and non functional requirements (e.g., delay, throughput, latency), efficient simulation and static formal analysis techniques have been defined in DIPLODOCUS and implemented in TTool.
The chapter is organized as follows. The related work section reviews several approaches for efficient design space exploration (section 3.2). DIPLODOCUS is then presented: the overall methodology is described, and main modeling capabilities are sketched(section 3.3). The semantics in LOTOS of the profile is then presented (section 3.4). Two other contributions are then addressed: the fast simulation of DIPLODOCUS models (sections 3.5 and

3.6) and the evaluation of the impact of shared resources (section 3.7). Main functionalities of TTool are then given (section 3.8). Conclusion of this chapter focuses on main results and future work (section 3.9).

3.2 Related Work

Design Space Exploration (DSE) of Systems-on-Chip is the process of analyzing various functionally equivalent implementation alternatives to select an optimal solution [188]. The most suitable design is commonly chosen based on metrics such as functionality, performance, cost, power, reliability, and flexibility. At system-level, DSE is challenging because the system design space is extremely large and so usual simulation-based analysis techniques fail to efficiently observe the above mentioned metrics. Contributions on DSE environments [40, 133, 53, 38, 166, 126, 186] generally rely on a high-level language to describe application functions and architectures. For example, [126, 166, 186] rely on UML or MARTE diagrams. Functions are sometimes described with only their cost [163]. Unfortunately, in many of these environments, architecture and application concerns are not independent [133], making the study of alternative solutions more complex. Second, they propose a way to map functions onto hardware execution nodes. Lastly, they introduce simulation techniques to simulate the system built from the mapping of functions over hardware nodes. But the level of abstraction being commonly rather low, simulation may also be slow. For example, [40] relies on an Instruction Set Simulator which executes the real code of the application. In [38], hardware components are considered at micro-architecture level, hence leading to long simulation times. Otherwise, other environments offer formal exploration, but generally limited to sub-elements of the platform [39].
SymTA/S [94] [97] and Real Time Calculus (RTC) rely on formal methods such as the real-time scheduling theory and deterministic queuing systems to determine characteristics of distributed systems. In SymTA/S the behavior of the environment is modeled by means of standard event arrival patterns including periodic and sporadic events with jitters or bursts. RTC imposes less restrictions by allowing deterministic event streams to be modeled with the aid of arrival curves denoting lower and upper bounds for event occurrences. 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. For that reason it may be tedious if not impossible to model tasks exhibiting a data dependent or irregular behavior.
[96] 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.
[187] provides means for formal and simulation based evaluation of UML/SysML models for performance analysis of Systems On Chip. UML Sequence diagrams are the starting point for the functional description. They are subsequently transformed into so-called communication dependency graphs (CDGs) which thus capture the control flow, synchronization dependencies and timing information. CDGs can be used as input to static analysis in order to determine key performance parameters like best case response times, worst case response times 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.
[141] presents a framework for computation and communication refinement for multiprocessor SoC Design. Stochastic automata networks represent the application behavior and the authors claim that this formalism allows for fast analytical performance evaluations. When it comes to mapping an application on an architecture, transitions and states have to be added to the application model. Hence, application and architecture matters and not strictly handled in an separated way. Due to a lack of data abstraction, the modeling of memory elements can quickly lead to state space explosion problem.
The PUMA [191] 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.
DIPLODOCUS offers a very clear separation between applications and architectures, and includes a high level of abstraction. Indeed, DIPLODOCUS is focused on control rather than on data, i.e., only abstract *samples* of data can be manipulated in the profile. Samples are untyped and carry no value: only their size is a relevant attribute. This high level of abstraction greatly reduces simulation times and makes formal proof techniques usable.

3.3 Basics of DIPLODOCUS

A UML profile customizes the UML language [151] for a given domain of systems. It may extend the UML meta-model, according to *semantic variation points*, and may provide a methodology. The DIPLODOCUS UML profile targets the modeling and Design Space Exploration of system-on-chip at a high level of abstraction [33]. The DIPLODOCUS methodology, depicted in Figure 3.1, comprises three main steps described below.

3.3.1 Application modeling

At first, the application is modeled using UML class and activity diagrams. Tasks are modeled as classes interconnected with *channels*, *events*, or *requests* to communicate. Data abstraction is a key point: channels do not convey values, but only a number of samples (data abstraction). Events are used for synchronization purpose, and requests are used to spawn tasks.
All tasks have a behavior modeled with UML-DIPLODOCUS activity diagrams. These diagrams come with usual control operators, complexity operators, operators to use communication medium (e.g., channels), and time operators. An extension has been proposed so as to model tasks definition and communications with a component approach, therefore facilitating the reuse of subparts of models, and the task parametrization [106].

3.3.2 Architecture modeling

Second, a candidate architecture is modeled in terms of interconnected hardware components (or nodes) using UML components placed in UML deployment diagrams. Three kinds of nodes are available in DIPLODOCUS: *Execution nodes* (CPUs), *Communication*

nodes (buses, bridges) and *Storage nodes* (e.g., memories). Each node has its own set of parameters such as *pipeline depth*, *miss-branching prediction rate*, *cache-miss rate*, etc. Also, all nodes - except buses - may be connected to one or several buses using UML links. For the time being, DMAs and hardware accelerators are represented by adequately parametrized CPU nodes.

3.3.3 Mapping

The mapping is meant to study whether a given architecture can execute a given application according to constraints. Application tasks and channels are meant to be distributed over hardware nodes. A UML deployment diagram is used for that purpose. A given task may be mapped only on one execution node. Channels may be mapped on paths built upon links, communications nodes, and storage nodes.
One of the strengths of DIPLODOCUS relies on its ability to perform simulation and formal verification both at application and mapping stages. Formal verification at mapping stage is further discussed in the next section. Although common functional properties are usually studied at application level, performance properties are investigated after mapping, e.g., resource sharing, that is, the scheduling on CPUs (can the architecture execute tasks on time), bus load (can a bus transmit the required amount of data), and properties related to power consumption and silicon area.

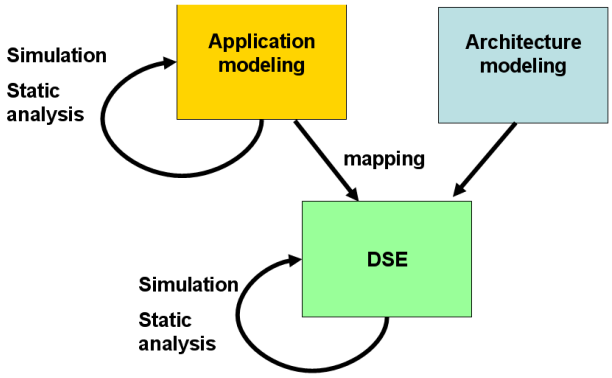


Figure 3.1: Methodology for Design Space Exploration

3.4 Formal support: LOTOS

DIPLODOCUS has been defined with formal verification in mind: formal verification for either the application model, or the mapping model. Indeed, the abstractions defined in DIPLODOCUS make formal verification accessible for complex systems, or at least subparts of these systems. Abstractions that do reduce the complexity are mostly the abstraction of data and complexity operators that can model a given computation as a time interval. Hardware components are also used at a very high level of abstraction. Those abstractions are further explained with the LOTOS semantics. While LOTOS has already been successfully experimented for property proofs on hardware [190], we propose its use to more generic platforms (SoCs).
The DIPLODOCUS semantics has first been briefly explained in [188, 33] and then further detailed in [121, 122]: the idea of this section is not to reproduce the full content

of those contributions, but rather to draw the general idea on how concepts defined in DIPLODOCUS can be expressed in formal languages. Emphasis is also put on differences between application-level and mapping-level semantics.
LOTOS [101] is an ISO-based Formal Description Technique for distributed system specification and design. A LOTOS specification, being itself a process, is structured into processes. A LOTOS process is a black box that communicates with its environment through gates using a multiway rendezvous offer. Values can be exchanged at synchronization time. That exchange can be mono- or bi-directional. LOTOS specifications may be formally verified with the CADP toolkit [85] using model-checking or reachability graph analysis techniques.
The semantics of DIPLODOCUS is also defined in UPPAAL, but only the LOTOS semantics is (briefly) presented in this document. UPPAAL [43] is based on a set of communicating automata. It explicitly supports synchronization actions between automata. Its strength also relies in clocks that offer convenient ways to model temporal behaviors - e.g., timers -. The UPPAAL toolkit offers simulation and model-checking capabilities.

3.4.1 Semantics at application level

3.4.1.1 Tasks operators

As previously described, a DIPLODOCUS application is composed of a set of communicating tasks. Operators used to describe task behavior are of four types:

- **Communication operators**: **read** from a channel, **write** a sample to a channel, **notify** an event, **wait** for an event, know whether an event has been sent (**notified**), **request** a task.
- **Control operators**: usual control operators, such as **variable modifications**, **loops** and **tests**.
- **Complexity operators**: operators to model a number of operations on integers (**EXECI**), floats (**EXECF**) or custom (**EXECC**).
- **Temporal operators**: operators to model deterministic and non-deterministic physical **delay**.

This set of operators makes it possible to describe applications' communications and algorithms whilst forcing the modeler to abstract data, thanks to channels that merely account for the amount of transmitted data.
The LOTOS semantics of all task operators is further described in Table 3.1, column "LOTOS Semantics before mapping".

3.4.1.2 Communications between tasks

Tasks communicate using channels, events and requests. While channels are used to model data stream between tasks - i.e., channels carry unvalued samples -, events and requests are meant to model synchronization schemes.
Three channel types have been defined: **BR-NBW** (Blocking Read - Non-Blocking Write, i.e., infinite FIFO), **BR-BW** (Blocking Read - Blocking Write, i.e., finite FIFO), **NBR-NBW** (Non-blocking Read - Non-Blocking Write, i.e., a set of elements, that is a memory).

Type	Task operators	LOTOS Semantics before mapping	LOTOS Semantics after mapping
Channel	Write n samples to a channel	n Write operations in FIFO, i.e., n times action on gate $wr\_\_ch$, see Figure 3.2	n cycles, and a request on a bus.
	Read n samples from a channel	n read operations from FIFO, i.e., n times action on gate $rd\_\_ch$, see Figure 3.2	n cycles and a request on a bus.
Event	Notify an event	Adds an event to the corresponding FIFO, i.e., performs an action on gate $notify\_\_evt$, see Figure 3.3	Same as before mapping.
	Wait for an event	Tries to get an event from a FIFO, i.e., performs an action on gate $wait\_\_evt$, see Figure 3.3	Same as before mapping.
	Notified	Returns the number of event in a FIFO using action $notified\_\_evt$, see Figure 3.3	Same as before mapping.
Request	Send a request (operator is called "request")	FIFO management is similar to the one used for events	Same as before mapping.
Control	loop, variable modifications, tests	Direct translation in LOTOS with corresponding LOTOS operators	Direct translation. Operators are executed in 0-cycle.
Complexity	EXECx n,m i.e., between n and m integer instructions	No semantics before mapping, i.e., this operator is ignored	The task executes between $n * perf$ and $m * perf$ cycles with perf being a constant value depending on the hardware performance on which the task is mapped.
Temporal	Delay $d_{min}d_{max}$ unit	No semantics before mapping, i.e., this operator is ignored	The task is blocked for Between n and m cycles with $n = d_{min} * frequency$ and $m = d_{max} * frequency$.

Table 3.1: Task operators

The LOTOS semantics of these channels is as follows: since channels convey no value, but only a number of samples, the two first channels can easily be translated into a simple process (see Figure 3.2) sharing a natural value (which represents the number of elements in the FIFO) between two processes using two gates: one gate to add a sample ($wr\_\_ch$), another one to remove a sample ($rd\_\_ch$). The last channel type (NBR-NBW) is also translated into a similar LOTOS process apart from the fact that no counter is necessary - since its is always possible to read and write -, and so no guards (// operator) are used before the actions on gates $wr\_\_ch$ and $rd\_\_ch$.

```
process ChannelBRBW__ch[rd__ch, wr__ch](samples:nat) : exit := (
[samples < 8] -> (wr__ch; ChannelBRBW__ch[rd__ch, wr__ch](samples + 1))
[]
[samples > 0] -> (rd__ch; ChannelBRBW__ch[rd__ch, wr__ch](samples - 1)) )
```

Figure 3.2: Application-level LOTOS semantics for a BR-BW channel

Events are meant to model synchronization between tasks. They can carry up to three parameters. Event communication semantics are the following: infinite FIFO, finite blocking FIFO, and non-blocking FIFO. The two first semantics have been selected because they reflect common synchronization schemes of embedded systems. The last one (Non-blocking finite FIFO) is particularly useful to model signal exchanges between tasks: indeed, software and hardware signals usually erase the previous one (e.g., Programmable Interrupt Controller, or UNIX signals). A separate LOTOS process accounts for each of the three semantics using the *Queue_nat* algebraic type. Figure 3.3 illustrates a non-blocking finite FIFO (the most complex case) for an event carrying only one natural parameter. Five cases have been taken into account:

1. The FIFO is not empty, and so, a *wait* action can be performed on the FIFO.
2. The FIFO is not full, and so, an event can be added to the FIFO (*notify*).
3. The FIFO is full, and so, an event can be added to the FIFO (*notify*) after the oldest one has been removed.
4. The FIFO is not empty, the *notified* action returns the value 1.
5. The FIFO is empty, the *notified* action returns the value 0.

Unlike channels and events which are one-to-one communications, **requests are many-to-one communications**. They rely on *n-to-one* infinite FIFO. The translation of requests is similar to the one of FIFO for events, apart from the fact that notification gates are instantiated n times, e.g., $notify\_i$ with $i \in 1 \ldots n$.

3.4.2 Semantics at mapping level

A mapping involves an application (i.e., a set of tasks and communications between those tasks), an architecture (i.e., a set of hardware nodes), a distribution of tasks onto hardware nodes (e.g., map the task *task1* onto the CPU *cpu1*), and a mapping of communication channels onto buses / memories. We have therefore defined a transformation function $tf()$ that takes as argument all above mentioned elements and generates a LOTOS specification

```
process Event__evt[notify__evt, wait__evt, notified__evt]
(fifo_1:Queue_nat, fifo_val_1:nat, nb:nat, maxs:nat) : exit :=
  [not (Empty (fifo_1))] -> wait__evt !First(fifo_1);
p_0_Event__evt[notify__evt, wait__evt, notified__evt](Dequeue(fifo_1),
fifo_val_1, nb-1, maxs)
  [] [nb<maxs] -> notify__evt ?fifo_val_1:nat;p_0_Event__evt[notify__evt, wait__evt,
notified__evt](
Enqueue(fifo_val_1, fifo_1), fifo_val_1, nb+1, maxs)
  [] [nb == maxs] -> notify__evt ?fifo_val_1:nat;p_0_Event__evt[notify__evt, wait__evt,
notified__evt](
Enqueue(fifo_val_1, Dequeue(fifo_1)), fifo_val_1, nb, maxs)
  [] [not (Empty (fifo_1))] -> notified__evt!1;p_0_Event__evt[notify__evt, wait__evt,
notified__evt](
fifo_1, fifo_val_1, nb, maxs)
  [] [Empty (fifo_1)] -> notified__evt!0;p_0_Event__evt[notify__evt, wait__evt,
notified__evt](
fifo_1, fifo_val_1, nb, maxs)
endproc
```

Figure 3.3: Application-level LOTOS semantics for a Non-blocking Finite FIFO

(see Figure 3.4).

3.4.2.1 Mapping issues

The mapping phase is meant to answer whether a system - i.e. an application mapped on a given architecture - satisfies a set of constraints, or not. More precisely, a mapping shall resolve contentions on shared resources (typically, a CPU, a bus, etc.) and therefore answer whether the computational and communication power offered by the architecture can execute the desired application, i.e., respect deadlines, etc. The LOTOS semantics is first defined with those issues in mind. As a consequence, the LOTOS specification of a mapping should take into account:

- The access control to shared resources, e.g., for tasks: access to CPUs, and for communication: access to buses. To take into account those accesses, we explicitly handle operating systems' scheduling policies as well as arbitration policies of buses.
- The time taken by tasks to execute operators, and the time taken by communications, e.g., bus and memory latencies.

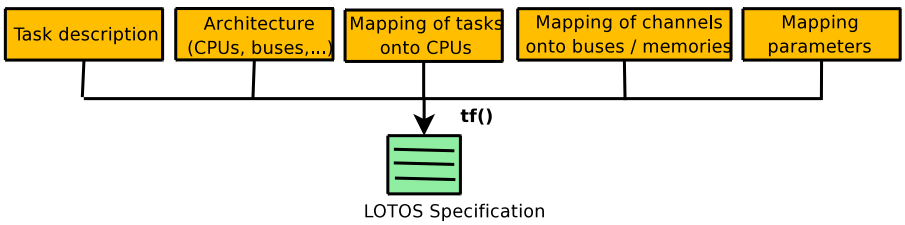


Figure 3.4: General approach

3.4.2.2 The Mapping-to-LOTOS transformation

All task operators and hardware nodes parameters are taken into account by the Mapping-to-LOTOS transformation ($tf()$). This function does not yet support hierarchical scheduling and virtual nodes [104] but there is no technical limitation to their integration.
Basically, the LOTOS specification is built upon four functional blocks:

- The **Scheduling manager** schedules tasks on each CPU. $tf()$ transforms each task into a state machine modeled in LOTOS: preemption can occur when a task is blocked in a state, but never when a task performs a transition from one state to another.
- The **Communication manager** handles channel-based communication between tasks running on the same CPU, or on different CPUs. Events and requests are assumed not to take communication resources. Indeed, the amount of data represented by those two synchronization features are assumed to be negligible with regards to channel-based communications. Similar assumptions were made for the simulation semantics [114] (which is less abstract and more tailored to simulation runtime issues).
- The **Task execution manager** handles operators to execute in each task that is transitions between task states.
- The **Clock manager** handles clock cycles on hardware nodes, i.e., it activates necessary hardware nodes when a new cycle begins.

The main process of the LOTOS specification works as follows:

1. At first, an initialization phase is used to settle various data structures, for each CPU (e.g., all tasks of a CPU are put in "ready" state), and for the communication manager: data structures related to channels, queues related to events, and so on.
2. A main loop on clock cycles is started: The system waits for the next tick (*tick* is a LOTOS action). Then, each CPU plus its operating system are considered one after another. Basically, a CPU is meant to interpret DIPLODOCUS application-level operators of the selected task:
   (a) Depending on its clock rate, the CPU is activated or not by the Clock manager.
   (b) If it is activated, then a first test is performed to see whether one task is in *running* state, or not.
   (c) If one task is in *running* state, then, the running state is activated from its former state. The task executes until either (i) it blocks (for example, it tries to receive one given event, and that event is not available): in that case, the scheduler is called, or (ii) it can perform an instruction consuming cycles (e.g., writing a sample to a non-full channel).
   (d) When the scheduler is called, it first checks whether at least one task is runnable. If no task is runnable, the CPU goes *idle*. Otherwise, a scheduling algorithm - implemented in LOTOS - is called to select another task. Then, the state machine of that task may be called, and so on.
3. Once all CPUs have been selected, a communication manager resolves all inter-CPUs communication, i.e., all communications set-up by tasks in previous cycle (i.e., all

read, *write*, *notify events*, etc.) are really performed only when all CPUs have terminated that cycle. This ensures (i) that a sample written on a CPU during a cycle may not be read by another CPU in the same cycle, and (ii) that the order of CPU evaluation has no impact on results.

The $tf()$ function may also generate debug information in the form of LOTOS actions performed at well-chosen points: actions to show scheduler data structures (e.g., list of runnable or blocked tasks), actions to monitor tasks states, actions to monitor the communication manager, etc.
Finally, $tf()$ has been defined with combinatory explosion in mind. Hence, $tf()$ tries to precompute possible synchronization between LOTOS processes: if possible, these synchronization are removed, and resulting processes put in sequence. Combinatory explosion may also be due to (i) non-deterministic elements: for example random, choice and temporal operators of tasks; (ii) Non-determinism in scheduling models: for example, in the round-robin scheduling policy, the possible indexes of tasks, in the tasks list. Abstraction is a key factor to reduce combinatory explosion. The main idea behind abstractions is to remove all software and hardware-related concerns that have no or little impact on evaluated properties (e.g., load on CPUs and buses). The next subsection is dedicated to abstractions.

3.4.3 Abstractions

3.4.3.1 Task abstraction (see Table 3.1, column "LOTOS Semantics after mapping")

- **Communication operators**. These operations are given a cost (in clock cycles), and are executed by the execution manager along with the communication manager, to make request on related buses. The cost in cycle depends of the hardware platform. For example, writing an 8-byte sample on a 32-bit processor takes two cycles. Also, the communication manager is involved for storing output samples, and for providing data to input operations. Note that these operations may be blocking, and so, the scheduling manager may also be involved.

- **Cost operators** are abstracted with a number of cycles depending on the hardware platform.

- **Other operators**: choice, loop, variable manipulation, etc. These operations are executed by the task execution manager. They take no cycle since there are used for control modeling purpose only, i.e., the execution cost in DIPLODOCUS is modeled only with *EXECx* operations.

- **Temporal operators**: They are abstracted with a number of cycles.

3.4.3.2 CPU abstractions

- **Parameters of CPU**: Data size (used for communication in channels), size of default integer and floating point data (used for EXEC operations), cost for each EXECx instructions, pipeline size (used for calculating the penalty induced by miss branch prediction), miss branching rate, data cache-miss ratio and penalty, time to enter/leave the idle mode, clock ratio.

- **The Operating System** is taken into account with scheduling algorithms (e.g., Preemptive priority-based, round-robin), switching time, synchronization management (events, requests) and communication delay (buffering for handling channels).

3.4.3.3 Communication abstractions (buses, memories)

Buses are meant to carry data samples with an arbitration policy between requests. The time a given transfer takes depends on the width of the bus. Bus arbitration is done on each cycle. Memory delays are modeled throughout bus latencies and cache-miss rates at CPU level, as proposed for the simulation semantics [114].

3.4.4 Semantics: discussion

The definition of the semantics focuses on the clear definition of DIPLODOCUS abstractions. We particularly described the semantic differences between pre and post mapping models. At application level, abstractions is mostly due to data abstraction, and operators to abstract computations. After mapping, abstractions rely mostly on how hardware components are defined with a basic behaviour and a set of carefully selected parameters.

3.5 Efficient and interactive simulations

Simulation is an interesting alternative whenever formal verifications cannot be successfully conducted, either because of combinatory explosions, or because of formal verification toolkit limitations.

3.5.1 Need for an efficient simulation

Simulation is intended to be used to evaluate - after mapping - the concurrency of operations performed on hardware nodes and more particularly on CPUs and buses. The first version of our simulation environment relied on the standard SystemC kernel and took advantage of the integrated discrete event simulator [188, 33]. In that simulator, a SystemC process is assigned to each active hardware node, and a global SystemC clock is used as a means of synchronization. Abstract communication and computation transactions defined in the application model are broken down to the corresponding number of wait cycles. Thus, all active components are run in lockstep.
Unfortunately, the inherent latency of the SystemC scheduler due to frequent task switches made the simulation suffer from low performance. However, simulation speed is an issue of concern as it represents one argument to justify accuracy penalties as a result of abstractions applied to the model. In order to achieve a better performance, a new simulation strategy leverages these abstractions by processing high-level instructions as a whole whenever possible. Furthermore, its simulation kernel incorporates a slim discrete event scheduler so that SystemC libraries are not necessary any more.

3.5.2 A new simulation engine

The simulation engine we have defined is based on transaction prediction. A transaction refers to a computation internal to a task, or a communication between tasks. Those transactions may obviously last from one to hundreds of clock cycles. Transaction durations are initially predicted according to the application model, that is to say their maximum

duration is given by the length of corresponding operators within tasks description. Unfortunately, a transaction may have to be broken down into several chunks of smaller size, just because for example, a bus is not accessible and so the task is put on I/O wait on its CPU. That transaction cutting due to prediction failure is more likely to happen when the amount of inter task communication is high and hence the need for synchronization arises. Unlike a conventional simulation strategy where all tasks are running in lockstep, a local simulation clock is assigned to each active hardware component. Thus, simulation granularity automatically adapts to application requirements as abstract measures for computational, and communicational costs of operations are specified within the application model. The coarse granularity of the high level description is exploited in order to increase simulation speed.
A problem at stake when considering transaction-based simulation is the scheduling of transactions themselves. This procedure embraces mainly four elements: tasks, CPUs, buses and the main scheduler. The sequence of the entities of the aforementioned list also reflects their hierarchy during the scheduling process: tasks are settled at the top layer and the main scheduler constitutes the lowermost layer. Based on the knowledge of their internal behavior, tasks are able to determine the next operation which has to be executed within their scope. This operation is encapsulated subsequently in a transaction data structure and forwarded consecutively to hardware components being in charge of its execution.
The basic idea is that a transaction carries timing information taken into account by hardware components to update their internal clock. Moreover, a hardware component may delay transactions and modify their duration according to the execution time needed by that specific component. By doing so, the simulation algorithm accounts for the speed of CPUs, the data rate of buses, bus contention and other parameters characterizing the hardware configuration.

The simulation algorithm is structured such that each command has to be prepared before it is able to execute. Simulation is carried out as follows:

1. At first, the preparation phase of the the first command of all tasks is entered.

2. Schedulers of all CPUs are invoked to determine the next transaction. Each CPU may have its own scheduling algorithm.

3. CPU schedulers register their current transaction at the appropriate bus in case the transaction needs bus access.

4. The main scheduler is in charge of identifying the runnable transaction t_1 having the least end time. Therefore, the main scheduler queries all CPUs which may check in turn if bus access was granted for their communication transaction. Bus scheduling is triggered in a lazy manner when the scheduling decision is required by CPUs.

5. The execution phase for transaction t_1 is entered.

6. If transaction t_1 makes another task T runnable which is in turn able to execute a new transaction t_3, transaction t_2 currently running on the CPU on which T is mapped is truncated at the end time of transaction t_1. The execution phase t_2 is entered. During the next scheduling round, the scheduler of the aforementioned CPU is able to decide whether to switch to t_3 or to resume t_2.

7. Commands which have been executing are prepared.
8. CPUs which have carried out a transaction are rescheduled.
9. Return to step 3

Two phases are more particularly at stake in that simulation procedure:

- **Preparation phase**. The current command must be checked for its total completion. If completed, the progress of the command is equal to its virtual length and the following command is prepared: A new transaction object is instantiated, its virtual length evaluated, and task variables might be evaluated or modified. Otherwise, the remaining virtual execution units is calculated. At the end of the prepare phase, transactions which might be processed by CPUs are known, only bus scheduling decisions are pending. That way, the prepare phase is crucial for the causality of the simulation.
- **Execution phase**. The objective of the execution phase is to update state variables of the simulation after a transaction has been carried out. In particular, channel states are updated, and transaction requests are added to CPUs and buses.

3.5.3 Experimental results

Several case studies have been considered to evaluate the performance of our simulation engine [114, 120]. Basically, the same systems have been simulated with the SystemC simulator, and with our simulation engine. The measurements have been performed under the following conditions:

1. Output capabilities of both simulators are disabled in order to measure the pure simulation time. No trace files are created.
2. The time consumed by the initialization procedure of both simulation environments is not taken into account for the same reason.
3. All measurements are subject to noise caused by the multi tasking operation system and interfering tasks running on the same CPU. Therefore, the average of several measured values has been taken into account for all different tests. Thus the noise should distort similarly the results for both simulators.

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. 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. [120] 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. 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 a factor of 10** in our testbed environment [120].

3.5.4 Interactive simulation

The above mentioned simulation environment allows for an interactive exploration of the application which is mapped onto a particular architecture. Simulation interactivity was first presented as a work in progress [115], then as regular papers [118] and at last demonstrated in several conferences [116, 30], including DATE [117]. Interactive simulation can be used as follows in TTool. 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 constraints of the metamodel, the next stage is to generate the C++ counterpart of the graphical 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 (in case the model should be verified) is required.

TTool encompasses a graphical interface to control the simulation and thus unburdens the user from familiarizing with a low-level simulation language. The feedback from the simulation engine is exploited by the graphical user interface and used to animate UML application diagrams. Most model-based code generation do have have such advanced simulation-to-model correspondence. For instance, the current command of a task is highlighted for following simulation progress on each task (see Figure 3.5). Different flavors of execution commands:

- A given amount of transactions, commands or time units can be simulated and the simulation may be interrupted when a particular hardware element (CPU, bus, bridge, memory) or an application entity (channel) processes a transaction.
- Save and restore the simulation state, especially useful when several branches of control flow are to be looked into.
- Reset the simulation to the initial state.

Simulation traces may be provided in several formats: the text based format is a simple listing of all transactions encountered on a given 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), CPU states (executing, idle, sleep mode). For debugging purposes of small designs, a user friendly HTML output may give an insight into the application's behavior. Transactions are represented on a time line for each hardware

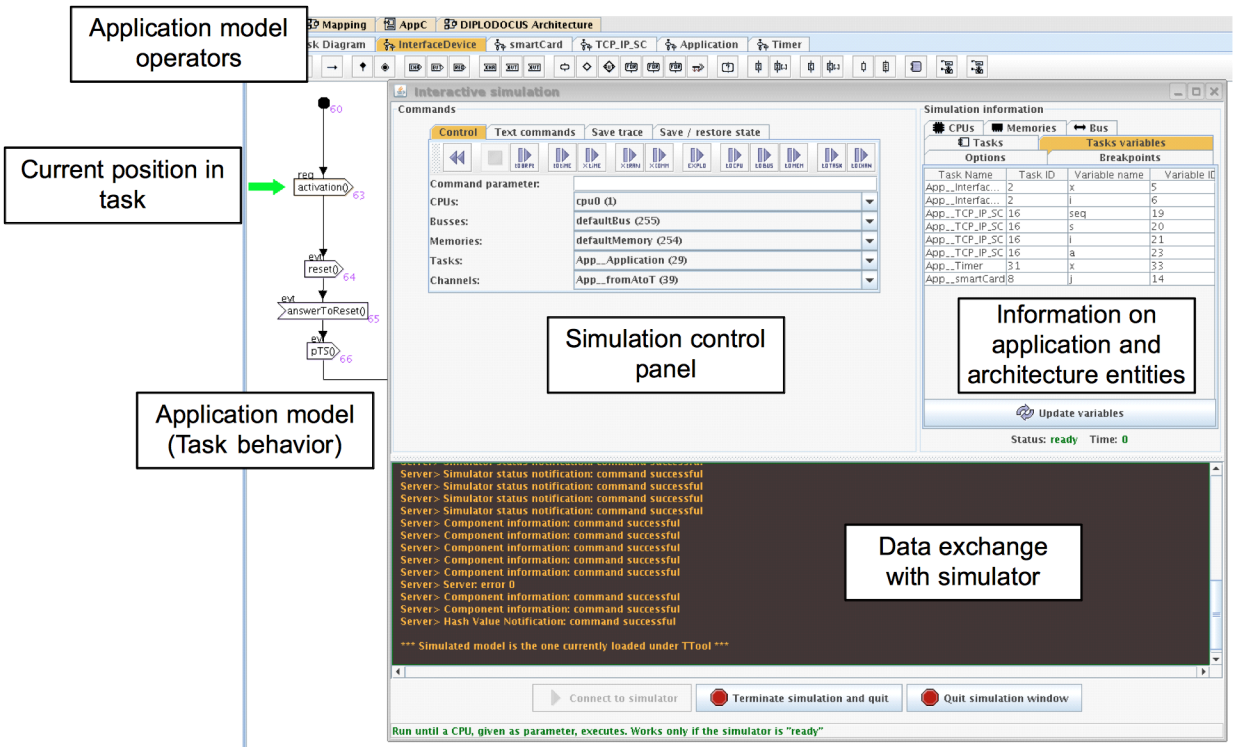


Figure 3.5: Interactive simulation with TTool

component and colored according to the task they belong to. In addition to traces, simulation results include performance figures 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. Some of those information are directly drawn on UML diagrams. For example, the load of each CPU is written on the upper right corner of the corresponding UML CPU node.

Briefly, interactivity is implemented as follows. 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 for the sake of performance. To get a better understanding of this interaction, let us now follow a user request which aims at defining 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 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. As stated previously, conditional breakpoints are intended to stop the simulation as soon as a condition (a function of task variables) evaluates to true. To deal with this kind of breakpoints, the simulator has to generate a C++ routine first which is subsequently compiled and attached to the process in the form of a dynamic library. This procedure prevents the cumbersome and costly interpretation of conditions at simulation runtime.

3.6 Coverage enhanced simulation

3.6.1 Rationale

To better see what we mean by *coverage enhance simulation*, let us consider the following example: an algorithm has 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 is crucial to try out both alternatives. Hence, the coverage of the simulation should be enhanced. 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 which branch he means to explore. 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.
A mean to automate this process is for example to allow the designer to precise for which choices he/she wants to make a full exploration of branches, or not. Another solution is to offer designers a way to ask the simulator to explore a given percentages of all branches or commands. A low percentage may result in exploring only one branch. Reciprocally, 100 percent coverage means the full coverage of the application. We have followed the second approach, that is, **making it possible for the designer to explore a given percentage of its model** (percentage of commands, percentage of branches). The simulator has thus been enhanced with formal verification capabilities: states of the simulator can be compared to previously explored states, in order to merge states and to derive a reachability graph [120]. This also resolves another issue. Indeed, when using LOTOS or UPPAAL, we experienced that the transformation of sophisticated mapping models results in complex syntactical structures, often pushing both UPPAAL [43] and CADP [85] model checkers to their limits. Since our simulation engine is dedicated to DIPLODOCUS mapping models, the code generator for that engine produces only manageable code.

3.6.2 Related work

There has been several efforts to extend the insight provided by simulation in order to address the changing behavior and demands of today's embedded applications. To this

end, simulation techniques are often combined with formal methods. The common ground of all approaches is the attempt to examine a large fraction of possible system executions, optimally the whole state space whilst limiting verification run time and memory consumption. The listed work differs from ours in the way concerns (application, architecture) are separated [49] [98], the abstraction level of the task model is chosen [182] [187] [139], and the model of computation emphasizes data or control flow [127].
In SESAME [182], the authors assume application behavior to be independent of timing and scheduling decisions.
Other approaches like [49] and [127] combine simulation with the theory of scheduling and shared resource allocation. Emphasis is rather put on task scheduling on processor-based systems and therefore separate models for architecture and application are not supported. In the second effort control flow within tasks is abstracted, the latter being considered as sources and sinks of event streams. This formalization is rather data flow oriented, whereas DIPLODOCUS places emphasis on control intensive parts of applications.
[187] suggests to use static techniques together with simulation of UML/SysML models for performance analysis of SoCs. A drawback of this approach is that data flow independence has to be kept, thus preventing more complex control flow primitives (distinctions, loops) with variable bounds to be part of the application model.
[139] introduces a formal executable system model based on communicating tasks interconnected with FIFO channels. As opposed to our approach, tasks are considered to be atomic and they are abstracted to best/worst case execution times. [98] introduces a methodology combining Model Checking and SystemC simulation. However, as compared to SystemC, DIPLODOCUS pushes abstractions further by enforcing the separation of application and architecture that way providing means for efficient Design Space Exploration. A more complete related work on simulation combined with formal verification techniques is available in [120].

3.6.3 Contribution

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). The state space within a task is defined by its local variables and the progress of the current transaction. Basically, the state of channels can be characterized by the number of samples currently stored in them. Each 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.

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.

Techniques have also been defined 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. Two techniques have been used for that purpose: **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, i.e. regardless of the underlying execution platform. Optimizations of this kind have been around for a while. However, Local Input Dependence Analysis is somewhat particular to the DIPLODOCUS model of computation. 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. All technical details are provided in [120].

3.6.4 Implementation issues

Three main enhancement have been made to the DIPLODOCUS simulator in order to support variable coverage features.

- Static analysis of DIPLODOCUS model. A bit vector is used to store dependencies. It is taken as input by the simulation engine.
- Exhaustive and coverage driven Simulation. Each command is enhanced with a *indeterminism* field. The simulation goes on until either the simulation stops, or a non deterministic transaction is met, or an already encountered state is reached. Efficient state hashing functions are split among tasks, channels, events and requests.

Implementing all this had an impact on the simulation speed. Experimental results are provided in [120]. Demonstrations of these new capabilities have been demonstrated in DATE'2011 [119].

3.7 Shared Resources Issues

3.7.1 Rationale

Handling concurrency in modern embedded systems has become a factor of utmost importance to ensure correct Quality of Service. For instance, multimedia applications in mobile devices can execute concurrently with radio-related applications (e.g., telecommunication protocols). These applications may have specific scheduling requirements (soft real time, intensive data transfer or execution loop, etc); furthermore applying one access policy to all applications is not the optimal solution [172]. Therefore, DIPLODOCUS was enhanced with capabilities to evaluate the influence of shared resources on a system's performance metrics such as latency, throughput and resources utilization: this enhancement relies on

the concept of virtual nodes [105]. Our methodology also enables the modeling of interactions of the system with its environment [105]. At last, we developed a specific SystemC-based simulation environment to monitor and analyze designed models [104]. This work is thoroughfully presented in [106], and was conducted with Freescale. The development of a new simulator is justified by the fact that it was necessary to co-simulate existing SystemC components along with the DIPLODOCUS models, at the cost of a slower simulation speed with regards to the one of the predictive C++ simulation engine (see section 3.5).

3.7.2 Related work

The back annotation techniques like MESH [49] and the one proposed in Schnerr et al. [168] focus on the modeling of task scheduling and extract contention attributes related to communication and memories from low level simulations. They rely on analytical and simulation techniques to estimate shared resources contention. Final code is used to estimate the performance. On the contrary, our methodology is applied early in the design flow, and so before far before the code is released.
On the other hand, early architecture exploration methodologies like Sesame [159] offer a clear distinction between application and architecture concerns, and facilitate flexible system-level performance evaluation. So far Sesame only provides schedulers to allocate computation resources to the application processes: it does not model communication architecture arbitration nor memory mapping.
Kempf et al. [110] present a simulation framework for MP-SoC platforms. They use a virtual processing unit (VPU) to schedule the execution of tasks mapped to a processor. The main difference with our approach is that we generalize the notion of virtual nodes to model access policies to any type of architecture resources, and that we are able to extract performance result of any shared resource.
Hierarchical scheduling methodologies for processors [136] or bus [143] try to optimize resources sharing between multiple groups with different scheduling requirements. Our approach applies the hierarchical control to all shared resources, and more importantly at a high level of abstraction.

3.7.3 Contributions

The contribution relies on two main elements:

- The **definition of Virtual Nodes** [105, 106]. A virtual node ensures - given a parameterized policy - the scheduling of accesses to shared resources (e.g. CPUs, buses, memories). The virtual node concept helps to build a well-structured simulation model, and also facilitates the creation of simple and reusable architecture component models. More precisely, a VN allocates the controlled resource to a requester, for example the VN of a CPU allocates the CPU to a task that is ready to execute, or the VN of a bus allocates the bus bandwidth to a CPU that is trying to reach the memory or other architecture nodes that are connected to the bus. A request is generated by a requester to access to a resource. It specifies the resource amount that the requester needs. For example, a storage request shall specify the size of data to transfer. In addition, a request has a priority in the case when the VN's access policy is priority based. VNs can be combined so as to specify hierarchical request handling (e.g., hierarchical CPU scheduling).

- **Extraction of contentions information** from simulation traces [104, 106]. Hardware architecture resources are instantiated from a library of pre-defined abstract models for architecture nodes that can be customized by setting the appropriate performance parameters (e.g. pipeline of a CPU, etc.), thus reducing the modeling effort. The designer can also use predefined access policies (with or without preemption): round-robin, fixed priority based, time slice scheduling, first-come-first-served. In addition, new access policies can be easily defined. Simulations produce VCD waveforms containing temporal characteristics of the analyzed system, i.e. of the application, the architecture and of the VNs. In order to get a global view of the system, the simulator provides, for each resource, the utilization factor and the average contention delay on each resource thanks to add-on observers. Buffer overflow situations on storage nodes are also indicated. At last, application temporal behaviors are summarized in terms of end-to- end latency of the application, tasks execution time, and the ratio of a task being ready or waiting to be scheduled by the VN of a computation node. Thus, the designer can make design decisions based on simulation results: for example, he/she can evaluate the access policies of shared resources, tasks memory mapping and the optimal capacity of resources (CPU frequency, memory size and hierarchy, bus speed ...).

3.8 Tooling

TTool [34] has been presented in previous chapter 2 as a toolkit for supporting the TURTLE profile. TTool also implement the DIPLODOCUS profile. In particular, the following capabilities are supported:

- Modeling of application, architecture and mapping stages in a UML way, or in a text-based format.
- Transformation of application, architecture and mapping models into LOTOS, UPPAAL or C++ format (press-button approach).
- Formal verification can be conducted from TTool, and results can be visualized directly in TTool. Reachability graphs can also be manipulated directly from TTool.
- Interactive simulations can be driven from TTool. Application and architecture models can be interactively animated at simulation time.
- Design space exploration can automatically be conducted using a scripting language defined in TTool. That is, many different mappings can be automatically explored (e.g., from 1 to n CPUs, etc.), and overall results are presented in a synthesis way.

The DIPLODOCUS implementation now represents a high fraction of the source code of TTool. TTool's website explains the ins and outs of DIPLODOCUS, provides all necessary information to install and configure TTool for DIPLODOCUS, and provides example models. An Android version will be available in the incoming years.

3.9 Results and perspectives

The DIPLODOCUS environment addresses the system-level design space exploration of Systems-on-Chip. DIPLODOCUS relies on a high-level language (UML), offers advanced

abstractions that make it possible to perform formal verifications or very fast simulations over models, and is fully supported with an open-source toolkit (TTool). Its coverage-enhanced simulator is an important contribution offering a real advantage between simulation traces and exhaustive analysis.
DIPLODOCUS has been successfully tested on many models (telecommunication, signal processing, multimedia, automotive systems), in the scope of academic and industrial partnerships, e.g., [106]. Two Ph.D. thesis have been successfully defended on DIPLODOCUS abstractions and simulation techniques [106, 120], and several student projects have contributed to either testing TTool, enhancing TTool, or working on specific issues.
DIPLODOCUS shall be enhanced in the next years with additional capabilities. In particular, a Ph.D. thesis is studying how to efficiently integrate the power consumption as a new partitioning metric. This includes the abstraction of advanced power management techniques. Requirement handling and property expression and verification is being addressed, and most recent work on that topics are detailed in chapter 4.

Chapter 4

Requirements and Properties

4.1 Context

The two previous chapters are dedicated to two UML profiles: *TURTLE* for the modeling and formal verification of time constrained systems, and *DIPLODOCUS* for the design space exploration of Systems-on-Chip. One of the strength of both profiles is the possibility to automatically derive formal specifications from UML diagrams. Formal verification refers to the study of whether a set of properties are satisfied in a system under evaluation. Unfortunately, in both profiles, only the system itself can be modeled, and not related properties. In other words, properties must be captured using other formalisms, e.g., CTL, thus limiting the TURTLE and DIPLODOCUS press-button approach for formal verification: the designer has to manually enter properties in a format that is related to the underlying formalism, and definitely not the one used for models (e.g., UML). We have addressed this problematic as follows.
First, the possibility to express requirements in both profiles. Indeed, properties to be verified are commonly derived from requirements. In UML 2, functional requirements may be expressed by use-case diagrams. Non functional requirements remain impossible to express. The promoters of SysML - the OMG-based UML profile for system engineering [152] - took the problem into account and introduced the concept of Requirement Diagrams. A first contribution is therefore to allow the capture of requirements in TURTLE using (enhanced) SysML requirements diagrams.
Second, the possibility to express properties to be satisfied by the model. We have proposed to formally define properties, and to link them to requirements. To do so, we have first proposed to model "basic" time-related properties with UML timing diagrams. TURTLE observers can directly be derived from those properties, and thus, the formal verification can fully be performed from UML models. Based on that first contribution, we have proposed to use another language for all kinds of properties - i.e., logical and temporal ones - based on SysML parametric diagrams: this new language is called TEPE, and is integrated to other profiles (e.g., DIPLODOCUS and AVATAR).
The chapter is organized as follows. First, requirement capture is presented in section 4.2. The modeling and verification of basic temporal properties is presented in sections 4.3 and 4.4, respectively. The diagram used for capturing these properties is called TRDD. The automated derivation of TURTLE observers from TRDD is also explained. The TEPE language is then presented in section 4.5: syntax, semantics, and main results are provided. Future work is presented in last section.

4.2 Requirement capture

4.2.1 Context and related work

A Requirement Diagram (RD) has a tree structure made of ≪requirement≫ nodes and ≪testcase≫ nodes. RDs use ≪derive≫ relations to express refinement hierarchy of requirements. Moreover, RDs use ≪verify≫ relations to link requirements to test-cases. Given one requirement R defined for one system S, one test case TC linked to a requirement R enables testing of S's implementation against R. In SysML the ≪verify≫ relation is meant to be used in the context of testing. In our approach, we use it for linking observers (i.e. verification guides, not test cases) and formal requirements. SysML syntax allows one to structure a set of requirements as a tree of ≪requirement≫ nodes.
Nevertheless, the SysML standard does not specify how to clearly state the nature of the requirement inside a ≪requirement≫ node. Indeed, requirements description is written in plain text and therefore remains totally informal. At first, we have enriched these requirements with temporal requirements, themselves enriched with a graphical and formal notation that we name '*Temporal Requirement Description Diagrams*', or TRDDs for short. Further, ≪derive≫ and ≪verify≫ relations have a formal semantics whenever formal temporal requirements are involved. This way, temporal requirements are formally linked to observers (SysML test cases) in charge of driving formal verifications.
Formal verification of UML models has been investigated by several research projects, in particular OMEGA [147] and ACCORD [88]. In both projects, requirements are captured outside the UML model. [134] proposes to express requirements in SysML Requirement Diagrams. [134] adds formality to temporal requirements and expresses the later using LTL formulas. Our first contribution relies on TRDDs instead. In [134] the system's structure is defined by a SysML block diagram. The blocks' behaviors are defined by Time Petri nets, as an alternative to SysML activity diagrams. The approach is partly automated by reuse of the TINA toolbox [44] and assumes the system's designer is capable of writing LTL formulas.
The importance of Requirement Diagrams has been acknowledged in the literature. Some authors take SysML Requirement Diagrams as they are without adding them any formality. Examples include the ModelicaML profile proposed by [2] for co-design of software and hardware and the profile defined by [183] for System-on-Chip. Conversely, our approach based on TRDDs adds formality to SysML requirement Diagrams as far as temporal requirements are concerned. We further bring solutions to achieve requirement traceability, an issue of prime importance as underlined by [183] and [7].

4.2.2 Contribution: TURTLE Requirement Diagrams

TURTLE Requirement Diagrams reproduce the tree structure of SysML Requirement Diagrams and introduce three types of nodes:

- A ≪requirement≫ node, e.g., *HybridPowerManagement* on Figure 4.1) defines an informal requirement expressed in plain text. The scope of the requirement is not limited. A requirement is defined by its name, its type (functional, non-functional) and its risk level (low, medium, high).
- ≪Formal Requirement≫ node (e.g. *F_Brake_HPMU* on Figure 4.1) formally expresses one temporal requirement. In general, one formal requirement is derived from an informal one. In Figure 4.1, *F_Brake_HPMU* is derived *Brake_HPMU*.

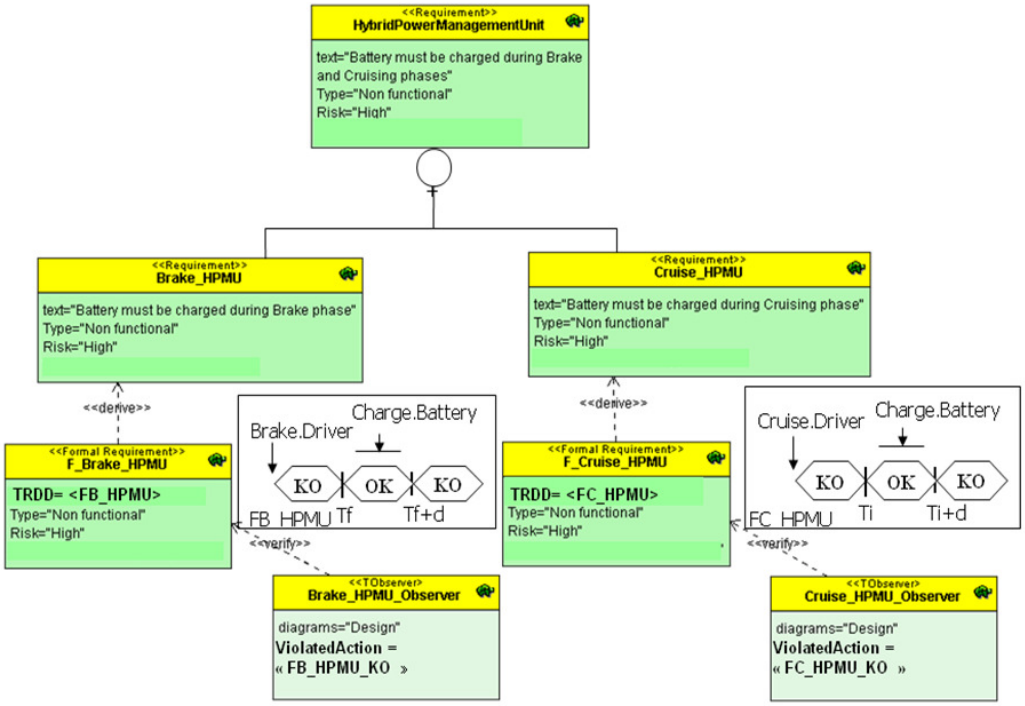


Figure 4.1: Example of a Requirement Diagram

All formal requirements are further described by a TRDD (Temporal Requirement Description Diagram, see subsequent sections). Therefore, a formal requirement definition include a name, a type, a risk level, and also a TRDD reference (see, e.g., $<TDRR>= FB\_HPMU$ on Figure 4.1).

- A ≪TObserver≫ node (e.g. *Brake_HPMU_Observer* on Figure 4.1) refers to one observer in charge of guiding formal verifications against one formal temporal requirement (i.e. one ≪Formal Requirement≫ node). Observers and formal requirements are linked with ≪verify≫ relations (In Figure 4.1, see, e.g., the ≪verify≫ relation from *Brake_HPMU_Observer* to *F_Brake_HPMU*). An observer is defined by a name, a type (design, analysis), the name of the diagram targeted by formal verification, and a *Violated_Action* attribute. Let us assume an observer OBS is linked to one formal requirement FR. Each time FR is violated during reachability analysis, OBS performs one synchronization action whose name equals the identifier given by the *Violated_Action* attribute of OBS. This way, FR's violation may be detected by inspection of the reachability graph transitions' labels.

Figure 4.2 defines the meta-model of TURTLE Requirement Diagrams. Informal and formal requirements are new stereotypes defined from SysML Requirements (*::SysML::Requirement* class). A formal requirement may be derived from an informal one (association labelled by *derive*), not the opposite. *TObserver* is a new stereotype defined from the SysML stereotype "TestCase" (*::TTDTestingProfile::* TestCase class). A test-case is an element used for requirement verification. Since an observer verifies one formal temporal requirement, it is connected to one formal requirement with a ≪verify≫ relation.

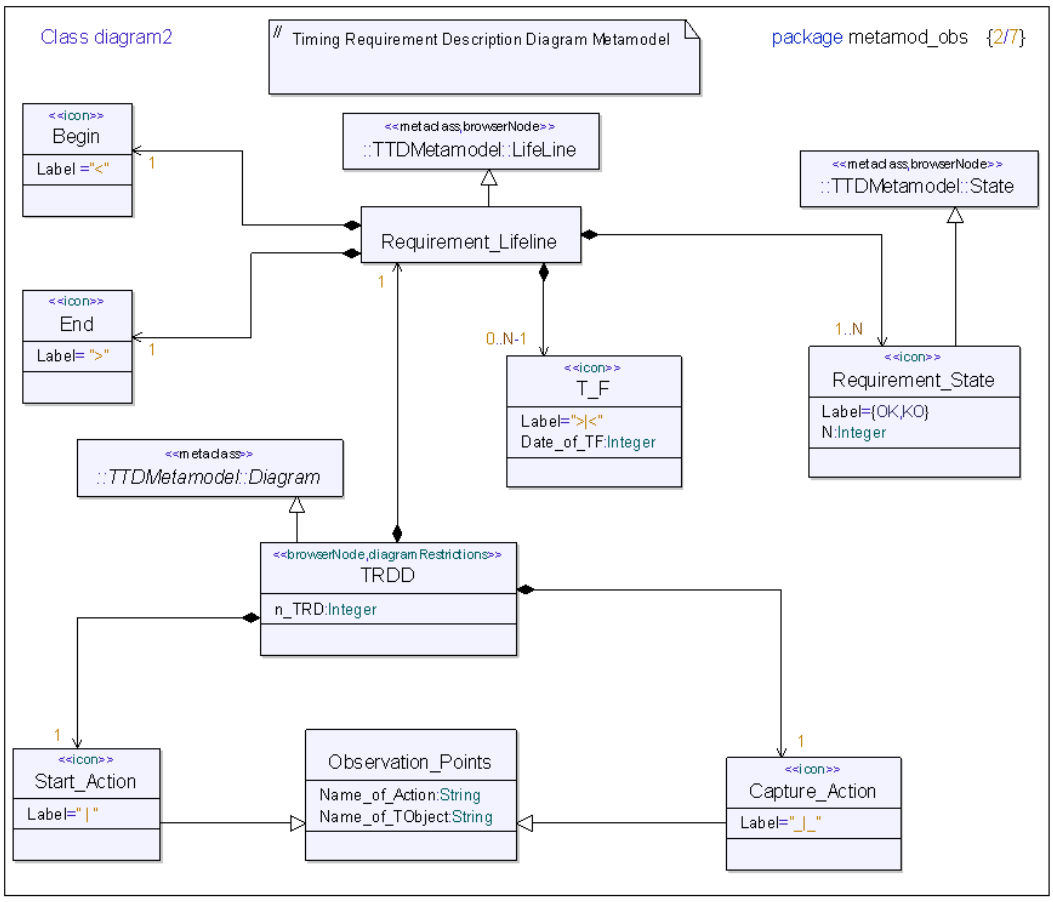


Figure 4.2: TURTLE Requirement Diagram Meta-Model

4.3 Modeling temporal properties with TRDDs

4.3.1 Related work and context

Temporal requirements may be expressed in a qualitative or a quantitative way. The former approach exclusively addresses the temporal ordering of events. The latter considers the order of events and temporal distances between events. Table 4.1 sets up a correspondence between temporal requirements and the bounded promptness properties defined in [7]. Notation: $S \models R$ stands for "system S satisfies requirement R".

Kind of TR	**Definition**
Promptness	R ensures that an event occurs before a deadline Tmax. $S \models R$ is true if that event occurs before Tmax.
Minimal Delay	R ensures that an event occurs after a minimum time Tmin. $S \models R$ is true if that event occurs after Tmin.
Punctuality	R ensures that an event occurs at one punctual date T. $S \models R$ is true if that event occurs at the T date.
Periodicity	R ensures that an event occurs regularly at modulo T dates. $S \models R$ is true if that event occurs at modulo T dates.
Interval Delay	R ensures that an event occurs between/outside a temporal interval $]Tmin; Tmax[$. $S \models R$ is true if that event occurs inside/outside temporal interval $]Tmin; Tmax[$.

Table 4.1: Temporal Requirements taxonomy based on [7]

In KAOS (Keep All Objective Satisfied [181]), requirements are expressed by logic formulas written in RT-LTL (Real Time Linear Temporal Logic). KAOS also includes a method for goal driven requirement elaboration. The KAOS tool Objectiver [148] allows analysts to elicit and specify requirements in a systematic way and to achieve traceability from requirements to goals. The interest of the KAOS methodology is to formalize and trace functional and non-functional requirements (including security, safety, accuracy, cost, performance) throughout the design cycle. Like KAOS, TURTLE handles formal requirements from capture and its toolkit contributes to achieve traceability.
Scenario based modelling techniques are also candidates for temporal requirement description. Timed Use Case Maps [103] (see TUCM in Table 4.2) describe Use Cases Interactions including absolute time with a master clock and relative time constraints (Duration, Timer). Visual Timed events Scenario [180] (see VTS in Table 4.2) represent events interactions. An event denotes an action which potentially occurs inside the system. VTS enables time representation. It may express partial orders and relative time constraints between events. Finally, Live Sequence Charts [60] (LSC in Table 4.2) extend Messages Sequence Charts (MSC) to represent scenarios. LSC enable distinction between possible and necessary scenarios.

TRDDs also reuse the concept of observation points introduced in VTS. Nevertheless, TRDDs do not implement a scenario paradigm. TRDDs rely on Timing Diagrams.
The ICOS toolbox uses one formalism [81] close to timing diagrams. Real Time Symbolic Timing Diagrams (RT-STD in Table 4.3) are applied to SoC design. Regular Timing Diagrams [7] (see RTD in Table 4.3) improve the situation: they enable representation of

Name	TUCM	VTS	LSC
Reference	[103]	[180]	[60]
Formal Language	Clocked Transition Systems	Timed Computation Tree Logic	Bűcchi Automata
Verification type	Model Checking	Model Checking (UPPAAL/Kronos)	Model Checking

Table 4.2: Scenario-based visual languages with formal semantics

Name	RT-STD	RTD	TRDD
Reference	[54]	[7]	Our contribution
Formal Language	Bűcchi Automata	Symbolic values	RT-LOTOS / UPPAAL
Verification type	Model Checking	Model Checking	Observers

Table 4.3: Visual Languages based on Timing Diagrams

partial order between diagrams.

4.3.2 Contribution: Definition of TRDDs

Figure 4.3 depicts the meta-model of TRDDs which relies on UML Timing Diagrams [16]. UML Timing diagrams are defined with two notations: the robust one, and the concise one. TRDDs extend the latter. Basically, a TRDD consists of one requirement lifeline and two observation points. The *Requirement_ Lifeline* class has several attributes: *Begin* and *End* symbols, N *Requirement States* (*OK* or *KO*) and N-1 *Temporal Frontiers*. *n_ TRD* gives the number of elements in the TRDD requirement description (*OK* or *KO*, also called "*requirement states*"). *Observation_ Points* refers to observation events, i.e. TURTLE actions used for observing system objects. For example, *Start_ Action* (resp. *Capture_ Action*) model a requirement capture ignition (resp. completion).

4.3.3 Example of a Temporal Requirement Description Diagram

The right part of Figure 4.4 depicts a TRDD modeling a process that must complete within 10 time units. The process is framed by two actions "*Start_ Process*" and "*End_ Process*" that we call "*observations points*". The latter are modeled above a TRDD *lifeline*. The diagram also includes a *temporal frontier* whose value equals 10 units of time. The temporal frontier distinguishes between two time periods - *OK* and *KO* - that correspond to requirement satisfaction and violation, respectively.

4.3.4 TRDD patterns

Three TRDD patterns have been defined in order to easily capture common situations, and still handle complex temporal ones. More precisely, a first pattern is dedicated to temporal constraints with one temporal frontier. A second pattern handles situations with two temporal frontiers. The third and last one supports any numbers of temporal frontiers.

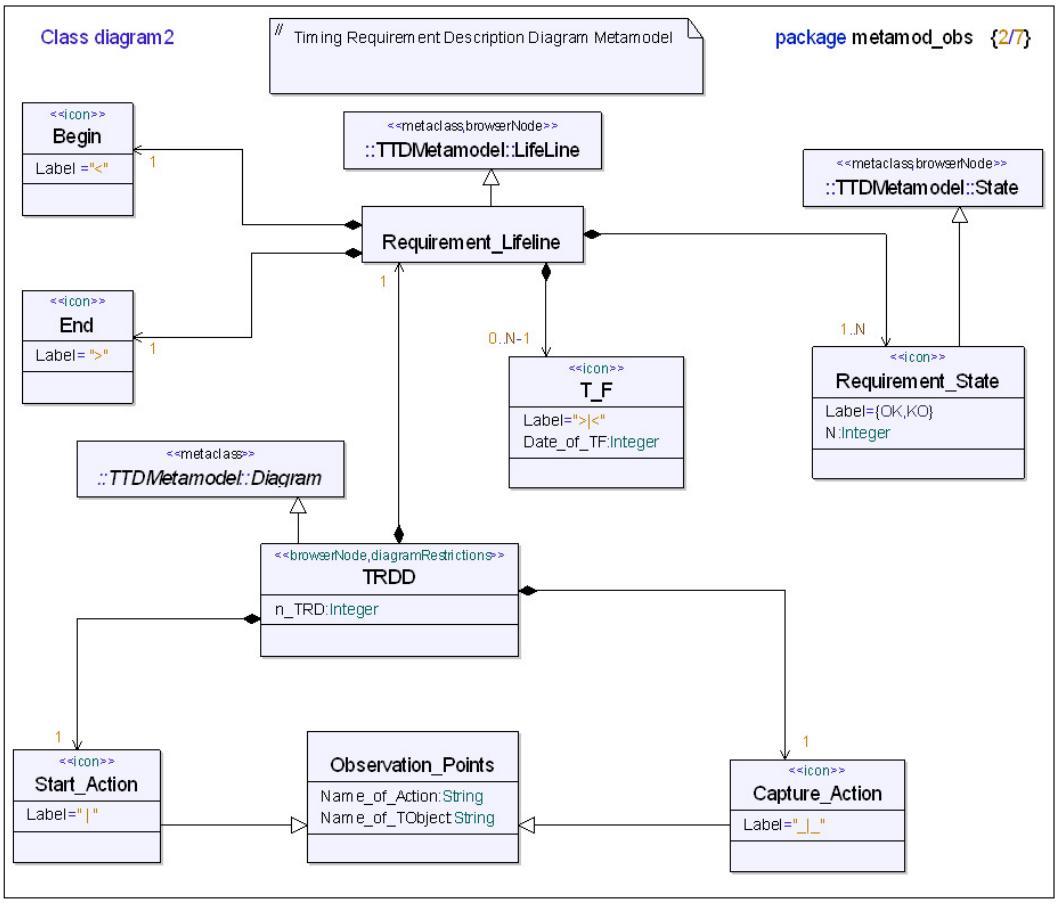


Figure 4.3: TRDD Metamodel

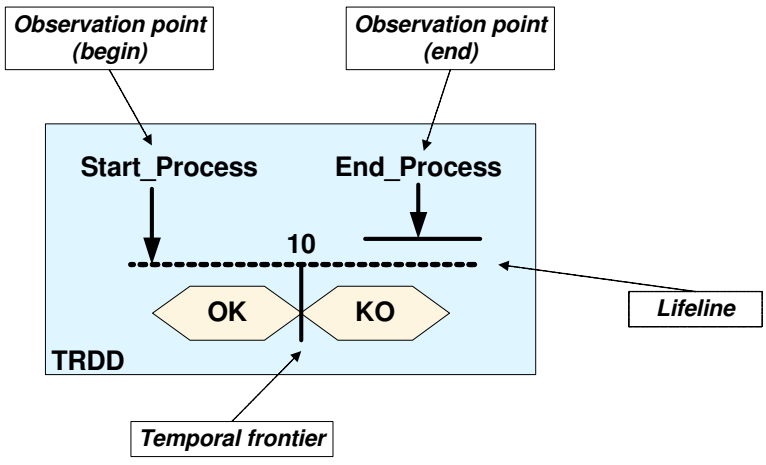


Figure 4.4: Example of a Temporal Requirement Description Diagram

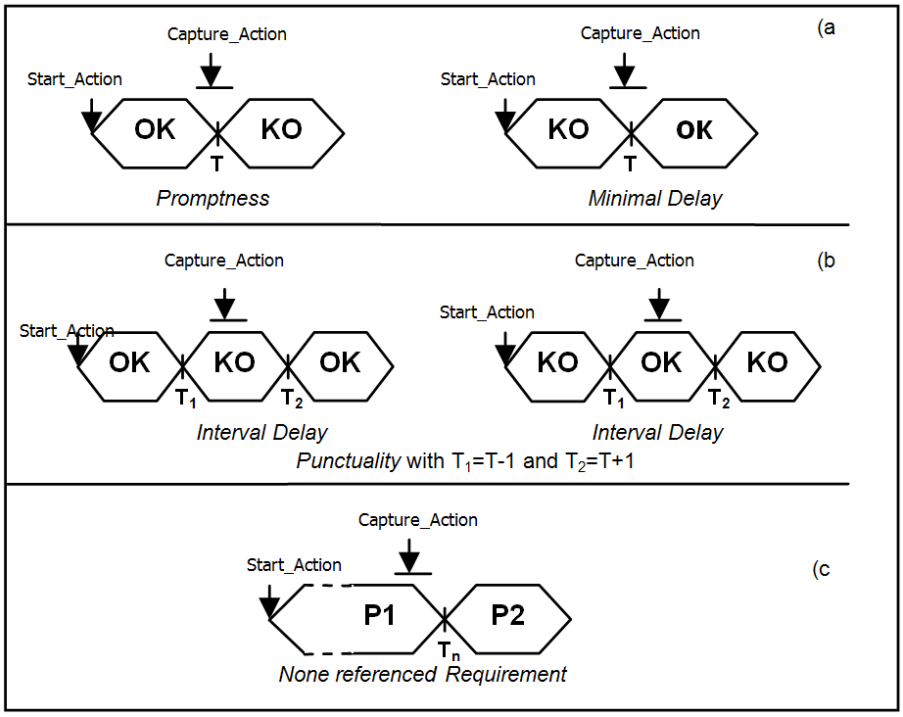


Figure 4.5: Comparison between TRDD patterns and the requirements presented in Table 4.1

Figure 4.5 identifies the three TRDD patterns and compares them with the taxonomy used in Table 4.1.

1. **TRDDs with one temporal frontier**. Corresponding events must occur before/after T time units. These requirements correspond to **Promptness** and **Minimal Delay requirements**, respectively.

2. **TRDDs with two temporal frontiers**. Corresponding events must occur between/outside interval $]T1, T2[$. These requirements correspond to **Interval Delay** and **Punctuality** requirements, respectively. A TRDD with two temporal frontiers $T1 = T - 1$ and $T2 = T+1$ corresponds to a Punctuality requirement on date T. Punctuality is verified on both sides of date T.

3. **TRDDs with N temporal frontiers and N+1 requirement states** (OK or KO). P1 and P2 represent two possible requirement states (OK or KO). Note: this kind of requirements is not referred in the taxonomy presented in Table 4.1.

4.3.5 Extending TRDDs

TRDDs presented so far may not model periodic events, bounded events and bursts. Periodic events happen every n time units. A bounded event may happen at any time, but its occurrence date is bounded by a time interval. Finally, burst events happen as grouped events with a maximum density per time interval. We have proposed extensions so as to model these schemes [79]. However, a more generic approach has been proposed in TEPE (see section 4.5).

4.4 Formal verification with TRDDs

4.4.1 Observer generation: rationale and related work

The three verification tools interfaced with TTool implement reachability analysis (i.e., RTL, CADP and UPPAAL). A reachability graph characterizes the states the system may enter from its initial state. Reachability analysis raises two issues related to the graph generation itself and to the graph's exploitation in order to decide whether a given requirement is met or not.
Dealing with graph generation, a first solution is to compute the entire graph. It raises two conditions. First, the system should be bounded, which means it should have a finite number of states. Second, sufficient computer resources should be available to generate and store the graph. When the two conditions are not met, partial reachability analysis is an alternative. For instance, model checkers such as CADP and UPPAAL implement on-the-fly model checking algorithms to detect property violations and satisfaction. TTool not only invokes external verification tools; it also provides verification results analyzers. In particular, it eases quick search of identifiers in large-size graphs. Moreover, TTool hides the complexity of formal verification by implementing a press-button approach: no knowledge of underlying formal languages is necessary. For instance, Well-known model-checking properties such as liveness or accessibility of some action A are answered by one single click on the UML action state of A.
Dealing with graph exploitation, we suggest transforming the graph into a Labeled Transition System (LTS). The latter's transitions are labeled with TURTLE synchronization action names. When two objects rendezvous, one transition is added to the graph and labeled by an identifier which unambiguously characterizes the rendezvous. Transitions no related to rendezvous are labeled by "nil". The information conveyed by the transition labels should allow one to decide whether requirements are met or not. TTool users may select a subset of rendezvous identifiers (i.e. transitions labels): other rendezvous transitions are subsequently labeled by "nil". Requirement satisfaction may then be decided from the result of searching for one or several transition labels. Also, the labeled graph may be minimized using Milner's observational equivalence. The resulting quotient automaton gives an abstract view of the system. This approach has often been used for communication architecture validation: the protocol layer is modeled in TURTLE and the reachability graph is minimized in such as way the quotient automaton characterizes the service rendered by the protocol layer.
Besides usual situations where two system's objects rendezvous, the post-generation analysis of the reachability graph particularly looks for rendezvous between the system's objects and other objects, termed as "observers", in charge of observing how the behavior of the system's objects influences on the satisfaction of the requirements to be verified. Let R be the requirement to be verified and OBS the observer which drives the verification against R. The gate list and behavior associated to OBS are hard to conceive since we expect OBS to remain non intrusive and to perform one synchronization action every time R is violated, so as that action appears as transition label in the reachability graph. Thus, OBS must synchronize with all objects whose behavior modifies R's satisfaction. Designing OBS by hand is therefore error-prone. We further expect OBS to remain non intrusive, the activity diagram of OBS is particularly complex to build. **We have thus experimented with the automatic derivation of observers from TRDDs** [80, 79].
The work on adding observers to a system model was pioneered by [107] for the ISO-based

formal description technique Estelle. The idea was to model the system's architecture and behavior in Estelle and to link the resulting model to a set of observers modeled using an Estelle-like language that grants access to all the variables, queues and timers of the modules included in the Estelle model of the system. The observer may output statistics on the simulation and stop or focus the simulation on particular points depending on the values of variables, queued messages and timers. Also, an observer may check the temporal ordering of events during one simulation run.
The simulation approach proposed by [107] for Estelle has been extended to verification and adapted to other modeling languages such as SDL and UML [52] [155] [72] [147].
In [107] and [72], observers are built up by hand. By contrast, [155] automates observers' construction. In particular, [155] uses CxUCC to link requirement expression to formal verification. Likewise, our contribution connects TRDDs to formal and observer-guided verification. The method we propose manages requirements before starting analysis. Conversely, [155] manages requirements during the analysis phase using context automata (termed as "use case charts"). An important difference is that TURTLE Requirement Diagrams enable description of both functional and non functional requirements, where use-cases exclusively address functional requirements.

4.4.2 Observer Synthesis and Traceability matrix

A TURTLE design defines a set of objects interacting inside a "closed world" that is impossible to access from the system's environment. The observer in charge of guiding formal verification against temporal requirements must therefore be integrated into the TURTLE model of the system. Thus, we give observers (i) access to the system's inner workings and (ii) the possibility to issue reports on any action that violates the temporal requirements whose TRDDs have served as starting point to create the observers. Unlike the observer-based simulation implemented by VEDA [107] for Estelle, the observer-guided verification implemented by TTool does not grant observers complete access to system's objects. Our solution relies on the following assumption: any change in the observed object OBJ that may be of interest to decide whether requirement R is met or not, should lead the observer OBS and OBJ to rendezvous. The activity diagrams of OBJ and OBS must be accordingly extended with relevant synchronization actions. OBJ and OBS rendezvous on paired gates. An association attributed by a *Synchro* operator links the OBJ and OBS objects in the TURTLE class diagram and an OCL formula states the two gates are paired.
A first option to design observers is to manually extend class and activity diagrams of the system's models with a set of synchronization actions having counterpart synchronization actions in the activity diagram of the observer: this is obviously error prone. We have thus defined a transformation algorithm that takes TRDDs and observers' description (identifier, observed diagram and violated action name) as input so as to automatically generate observers. That algorithm is implemented by TTool. It works with the intermediate language defined in TTool (TIF - TURTLE Intermediate Format), which means that observers are built directly in TIF and integrated into the TIF specification obtained from considered diagrams. TIF is an intermediate language used as TTool to have a common structure between UML diagrams and underlying formal languages. Basically, TIF is close to a textual form of TURTLE designs, that is, it is based on object decomposition, synchronous objet relations, and object behaviours described with activities. Since observers are directly generated in TIF, observed TURTLE diagrams remain unmodified. Additionally, observers are designed to be non-intrusive. More precisely, the behavior of observed

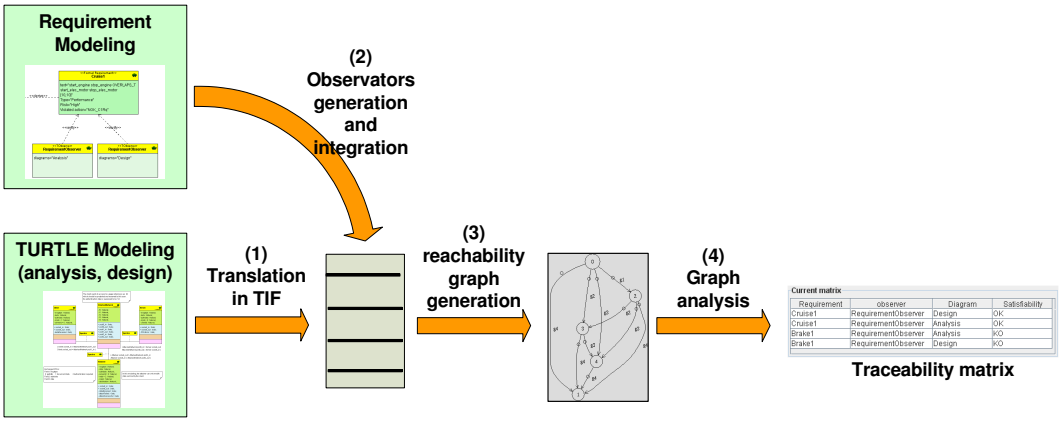


Figure 4.6: Observer-based formal verification procedure

objects is unmodified as long as low-level risk requirement are observed. Conversely, the verification process stops as soon as one high-level risk requirement is violated. In that latter case, observation remains non-intrusive as long as requirements are satisfied.
The fully automatic observer-guided verification process encompasses the following steps (see Figure 4.6):

1. TURTLE design diagrams are translated into a TIF specification, using the usual translation process.

2. For each observer linked to one formal requirement and referencing one design diagram, an observer is added to the TIF specification.

3. A reachability graph is generated using, e.g., the RTL verification tool.

4. The reachability graph is analyzed using model-checking techniques or minimized with respect to an equivalence relation and a set of events. The output of the model-checking or minimization process provides information to create a traceability matrix. For each requirement R listed in the matrix, it is indicated whether R is satisfied or not.

4.4.3 Observer generation

4.4.3.1 Observers

An automatically generated observer is a TURTLE class stereotyped by $\ll TObserver \gg$ (see Figure 4.7) and defined at the TIF level. Its name matches the name of the observer defined in the TURTLE Requirement Diagram. Its attributes refer to the dates defined in the temporal frontiers of the TRDD of the formal requirement verified by the observers. A TObserver also includes gates. Three types of gates are defined.

- **Gates to observe the model**. They correspond to the observation points defined in the TRDD of the observed requirement. Synchronizations are made possible by the Synchro relations that link the observer to the objects of the observed system. An observation gate observes either the *start* or the *capture* observation points (see [79])

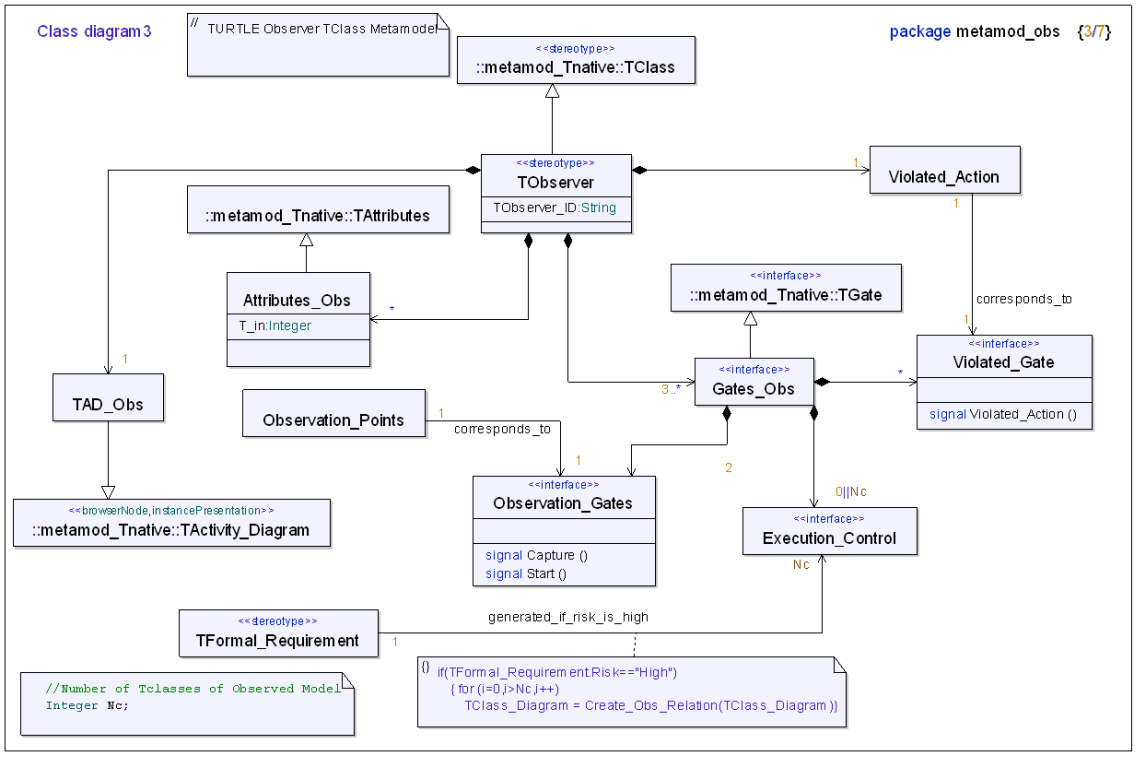


Figure 4.7: Meta-model of observers

- **Gates to trace the observed temporal and formal requirement**. These gates correspond to Violated_Action defined in observers.

- **Gates to control the system in which the observer is inserted**. They enable to stop the verification process when a high-level risk requirement is violated.

4.4.3.2 Observers behavior

Like any active TURTLE object, an observer's behavior is defined by one activity diagram (Figure 4.7). The latter is structured into three sub-parts: an anti-blocking system, the observation of the property itself, and the stoping of the system when a high-level risk property is violated. All these behaviours are thoroughfully explained in [79].

4.4.4 Traceability matrix

The reachability graph generated by RTL or CADP conveys information for creating a traceability matrix dedicated to temporal requirements. In practice, TTool seeks the reachability graph for transitions whose labels correspond to violated actions. Results of search for violated actions are collected in a traceability matrix. As suggested by Figure 4.8, a traceability matrix lists the temporal requirements declared in the TURTLE Requirement Diagrams of the system and assigns each requirement one verdict: OK if the requirement is met; KO otherwise. In Figure 4.8, formal requirements *F_ Brake_ HPMU* and *F_ Cruise_ HPMU* are violated (c.f. the *KO* labels in the Satisfiability column).

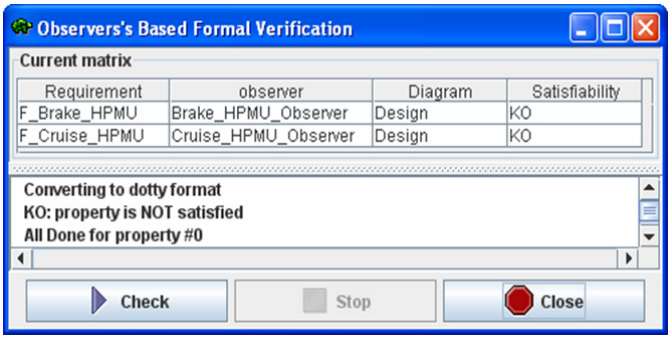


Figure 4.8: Traceability matrix of the HybridPowerManagementUnit requirement

4.5 A more generic approach to property modeling: TEPE

4.5.1 Context and related work

TRDDs have been succesfully used for the proof of basic temporal properties [79]. However, as explained in previous sections, they are totally dedicated to temporal situations (e.g., punctuality) and can definitely not be integrated in more complex properties (e.g., prove a punctionality property only when a given boolean variable is equal to "true"). The TEPE language has been introduced for that purpose: dealing with complex properties made of logical and temporal sub-properties.
The TEPE language 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 atomic actions carrying the same label. In this article, we use the latter meaning of *event* as a synonym for **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) 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. TEPE has been published for the first time at UMLFM'2010 [112] and fully defined in [120].

4.5.1.1 Property specification

System Verilog [5] provides concurrent assertions for describing behavior that spans over time. The underlying event model is based on clock ticks. TEPE constraints operate on physical or logical time.
The e-language [185] 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 logics and the expres-

siveness of its temporal layer resembles the System Verilog specification language. PSL is also tightly coupled to clock based events. 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 [150] 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.
[174] advocates a nice graphical notation which aims to simplify the formalization of requirements for model checking. System executions are expressed in the form of timeline 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.

4.5.1.2 Property specification in UML

The MARTE profile embraces the Value Specification Language VSL [153]. 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 [61] [95] 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) [140] [14], which describes system events of different types as abstract clocks. The language supports relationships between 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. Moreover, our objective was to suggest a concrete syntax (based on UML/SysML) paving the way for a seamless integration into various modeling environments (e.g., TURTLE, AVATAR, DIPLODOCUS, but also other UML profiles).

4.5.2 Contribution: TEPE is based on Parametric Diagrams

TEPE constraints directly refer to states, e.g. particular valuations of system attributes and also to signals. Model of Computations having one of two attributes 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 4.9, vertically cascaded TEPE constraints determine the activation period (AP) of the constraint immediately above. This is suggested by vertical *activation period* arrows. In turn, they only operate on signals and properties during the interval specified by the constraints immediately below. Properties are propagated along the vertical axis (cf. *property arrows* in the figure), named state axis. A constraint receives an input property to be verified and produces an output property, which may be the final result or subject to further verification. During its activation period, a constraint observes three signals represented by horizontal arrows in Figure 4.9. The signal $s1$ serves as **precondition** for the verification of the **liveness** of $s2$, and the **safety** of the system prohibiting the occurrence of $s3$ between $s1$ and $s2$. To account for time constraints, TEPE operators may specify a minimum (T_{min}) and maximum duration (T_{min}) of the interval bounded by $s1$ and $s2$, as formerly defined in TRDDs.
We have formalized with transfer functions (F_{prop}, F_{sig}, F_{act}) how incoming arrows in Figure 4.9 ($s1$, $s2$, $s3$, AP_{n-1}, P_{n+1}) relate to outgoing arrows (P_n, s_{out}, AP_n). Only main ideas of this semantics are given herefater, full details are nonetheless provided in [120]

Since SysML Parametric Diagrams (PDs) establish constraints between elements of the 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.

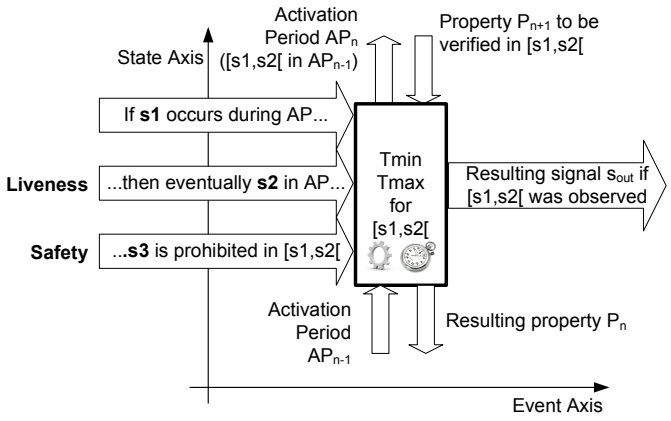


Figure 4.9: Intuition for the TEPE semantics

In TEPE, each property is expressed as a graph of TEPE Constraints: *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 TEPE meta model is depicted in Figure 4.10. All stereotypes of PDs are derived from their respective SysML counterpart: *Blocks*, *Constraints* and *Links*. A block provides a scope to *Attributes*, *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 a converse input/output configuration.

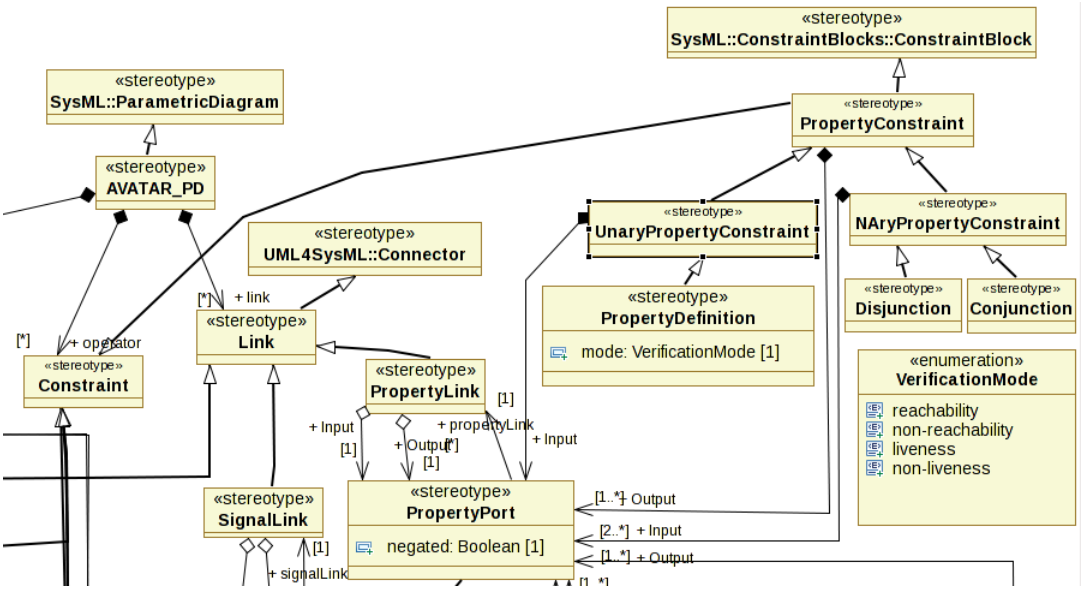


Figure 4.10: Excerpt from the TEPE PD Meta Model

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, properties can be split over several diagrams.

4.5.3 Example of a TEPE model

The example in Figure 4.11 informally presents operators of PDs, which are described in [120] in more detail. 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. An *Alias* constraint combines the two signals $s1$ and $s2$. The resulting signal corresponds to a logical disjunction and is thus raised upon occurrence of either $s1$ or $s2$. The two properties, established by the *LS* and *TC* constraint, are evaluated by an *AND* constraint. The *LS* constraint requires that upon occurrence of an $s3$ signal, the compound signal resulting from the *Alias* constraint must be observed as well, i.e. $s1$ or $s2$. Note that the negated input of the *LS* constraint is connected to the toggle signal of the equation. Therefore, if the value of z changes or 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 $s2$ to be occur *less than 10 time units after* signal $s1$. 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.

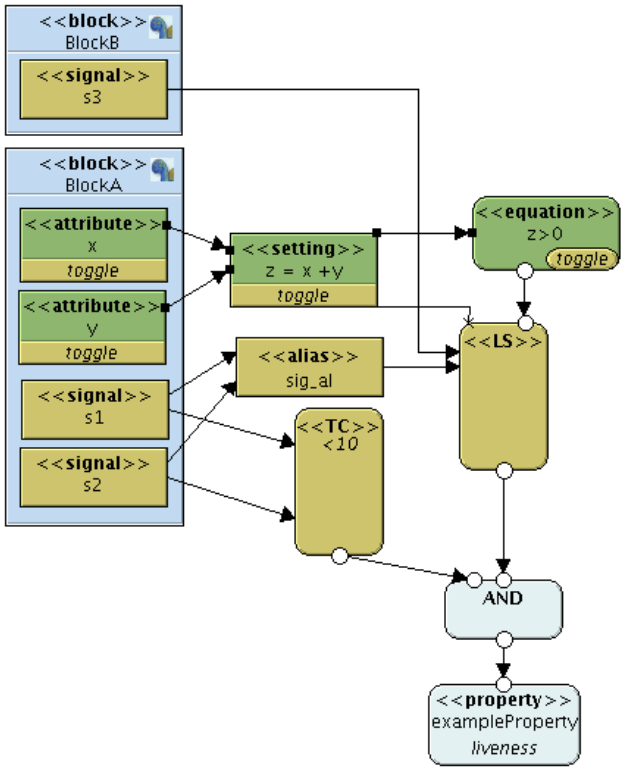


Figure 4.11: Example of a TEPE Parametric Diagram

4.5.4 TEPE semantics

The combination of Metric Temporal Logic (MTL) [125] and Fluent Linear Temporal Logic (FLTL) [132] is especially useful for reasoning about temporal properties in terms of both system states and events. They both serve as support for defining the semantics of TEPE operators [120].

Metric Temporal Logic (MTL) is presented in [125]. Basically, MTL 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 a 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 required to exhibit the properties of a metric.
Fluents are defined in [132] as "state predicates whose value 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.
A generic TEPE constraint is a nine tuple (F_i, f_{neg}, f_{out}, T_{mm}, F_{ai}, F_{ao}, $\mathbb{F}_{prop}$, $\mathbb{F}_{sig}$, $\mathbb{F}_{act}$) comprising

- A set of input fluents: $F_i = \{f_1 \dots f_n\}$, $0 \leq |F_i| \leq 2$
- At most one negated fluent: f_{neg}
- At most one output fluent: f_{out}
- A set of time parameters: T_{mm}, $0 \leq |T_{mm}| \leq 2$
- At most one pair of an activation period input and a property output: $F_{ai} = \{f_{n-1}, P_n\}$
- A set of pairs of an activation period output and a property input: $F_{ao} = \{\{f_{n,1}, P_{n+1,1}\} \dots \{f_{n,j}, P_{n+1,j}\}\}$, $0 \leq |F_{aio}| \leq 2$. To simplify matters, $f_{n,1}, P_{n+1,1}$ is written as f_n, P_{n+1} if there is exactly one pair.
- A transfer function for the property output if applicable $\mathbb{F}_{prop}$: $F_i \times f_{neg} \times \{P_{n+1,1} \dots P_{n+1,j}\} \times f_{n-1} \times T_{mm} \rightarrow P_n$
- A transfer function for the output fluent if applicable $\mathbb{F}_{sig} : F_i \times f_{n-1} \rightarrow f_{out}$
- A transfer function for the activation period output if applicable $\mathbb{F}_{act} : F_i \times f_{n-1} \rightarrow \{f_{n,1} \dots f_{n,j}\}$

As explained in [120], all TEPE constraints are built upon this generic definition.

4.6 Results and future work

Requirement modeling and property expression is a topic of utmost importance in semi-formal engineering. Requirement modeling in SysML is a quite recent topic, and property modeling is still an open issue, with various contributions in OCL, CCSL, UML timing diagrams, or TEPE. The first results we obtained with UML Timing requirements diagrams have been used as a basis to define a more generic property language suitable for both logical and temporal properties. The proofs of so-described properties with automatically generated observers is also part of our contributions.
Modeling TEPE diagrams is now possible in TTool. TEPE Diagrams can be used either in the scope of DIPLODOCUS [120] or AVATAR models [112] (AVATAR is defined in chapter 5). In the case of DIPLODOCUS, simple TEPE diagrams can be taken as input by the coverage-enhanced simulation engine defined in DIPLODOCUS, and so, formal verification of properties can be conducted taking as input DIPLODOCUS models and properties expressed in TEPE. Example of this is provided in [120].

Since TEPE definition is recent, the modeling of properties in several case studies will surely modify the current semantics. However, up to now, complex logical or temporal properties have been modeled in small-size diagrams. The full integration of TEPE in other profiles is still an open issue that shall be addressed in a near future. In particular, its relation with MARTE is also under study.

Chapter 5

AVATAR: Handling Safety and Security Issues in the Same Models

5.1 Context and problematic

For over ten years, the TURTLE UML profile has been used for modeling many different kinds of systems, e.g. in order to dimension these systems [32], to analyze protocols [22], to design embedded systems [35], and to deploy software components [36]. It has been used in several industrial and research contexts. Reviews from papers we submitted, partners' remarks, returns from users experience helped us to identify a few drawbacks. In particular:

- TURTLE was based on UML 1.4. Even if improvements were made progressively towards a UML2 flavor, the core of TURTLE remains built upon an old version of UML. For example, composite structure diagrams are not supported in TURTLE, thus leading to interconnect classes only at class diagram levels, without any notion of ports or parts. This problem was partially addressed in the CTTool profile [8].

- Another issue is the support of requirement and property capture, which was proposed to be made with SysML or UML2 diagrams [80, 113]: Thus, SysML diagrams are now mixed in TURTLE with UML 1.4 diagrams.

- TURTLE design behavior is based on activity diagrams. It is now very common that activity diagrams are used only in analysis stage, and state machines for describing the behaviour of components.

- Security is now a topic of utmost importance in embedded systems, in addition to safety issues. Reworking TURTLE was thus necessary to handle both safety and security proofs from the same UML and SysML models. A lot of work is concerned with modeling environments capturing safety properties during all development stages, including requirements engineering and design. To reduce verification complexity, formal proofs at design stage are usually applied to high-level models, or to subparts of more detailed designs. Unfortunately, current approaches are mostly dedicated to safety properties. And so, the developer has to come up with additional models relying on different formalisms dedicated to security proofs. Additional effort has therefore to be spent on the creation and maintenance of two models - one for safety issues, and one for security ones - instead of one, and also on model consistency.

Adding security to UML has already been proposed [108]. Unfortunately, the inherent structure of UML lead to solutions where security mechanisms and properties are mixed on the same diagrams. On the contrary, SysML clearly distinguishees between requirements and design over different views and diagrams.

For all these reasons, we decided to propose a new SysML-based environment - named AVATAR - and to integrate security properties and mechanisms in the core of this environment. AVATAR borrows from TURTLE many points, in particular formal proofs can be performed with a press-button approach in TTool. AVATAR supports the same methodological stages, with sometimes very similar modeling schemes (e.g., the analysis stage), or different ones (system architecture at design stage). The most interesting novelty is described in this chapter: the support of security properties and mechanisms. The new AVATAR methodology is first sketched in section 5.2. Design diagrams are then presented in section 5.3. Security support is described, from requirement capture stage until the formal verification stage (section 5.4). The prototyping stage has been deeply enhanced with regards to the TURTLE one: it is thus presented in section 5.5. At last, section 5.6 concludes this chapter.

5.2 A new SysML-based methodology

1. **Requirements**. Requirements of the system are first captured with SysML Requirements Diagrams. Requirements are organized in a tree-based fashion. Both safety and security requirements can be captured. Attacks can be modeled within Parametric Diagrams specifically customed to support attack trees.

2. **Analysis**. The analysis phase relies on sequence diagrams structured either with activity diagrams or interaction overview diagrams. This stage is very similar to the TURTLE's analysis stage.

3. **Design**. The general structure of the system is modeled with SysML block Diagrams. The behaviour of each block is described with a state machine.

4. **Simulation and formal verification**. A press-button approach makes it possible to perform simulations with model animation. Safety and security proofs can also be performed directly from the design models without prior knowledge about underlying formal verification techniques. Models can then be modified depending on verification results. Safety and security proofs rely on UPPAAL [43] and on ProVerif [46], respectively. Safety properties can be expressed in TEPE [112]. Security properties can be expressed within pragmas [158].

5. **Code generation and prototyping with SocLib**. The TTool code generator can output C/POSIX code from design models. The generated application code can then be compiled along with MutekH using the appropriate kernel configuration and cross-compiler to target a PowerPC based SoCLib platform. The SoCLib simulator can then be started and the code - generated and compiled during previous step - is loaded and executed like on real hardware. Debugging can be performed at two levels using both the GNU debugger, and simulations traces. Simulations traces can be displayed in TTool during code execution or later. These traces are displayed in

TTool under the form of UML sequence diagrams. AVATAR models can then be further modified as needed in order to generate a new code.

Simulation and formal proofs are meant to be executed during first iterations on the system model. On the contrary, the prototyping of the system is expected to be performed during the last iterations, that is, on more refined models. In all cases (simulation, verification and prototyping), results are directly displayed by TTool in a SysML fashion, therefore facilitating the identification of problems directly on SysML models.

5.3 AVATAR design

Apart from their formal semantics, AVATAR Block and State Machine Diagrams only have a few characteristics which differ from the SysML ones.
An AVATAR block defines a list of attributes, methods and signals. Signals can be sent over synchronous or asynchronous channels (TURTLE supports only synchronous communications). Channels are defined using connectors between ports. Those connectors contain a list of signal associations.
A block defining a data structure merely contains attributes. On the contrary, a block defined to model 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 delay 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 last from t_{min} to 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 main features for supporting real-time system schedulability analysis. The semantics of AVATAR has been first defined in UPPAAL, but the semantics of a subset of the profile has also been defined in the pi-calculus language supported by ProVerif [46]. This semantics is further explained in the next section.

5.4 Extending AVATAR for security purpose

5.4.1 Modeling and verifying embedded systems with security constraints

A wide range of methodologies and tools has been proposed for modeling and verifying security in embedded systems.
[177] proposes to verify cryptographic protocols with a probabilistic analysis approach. Protocols are represented as trees whose nodes capture knowledge whilst edges are assigned with transition probabilities. Although these trees could include malicious agents in order to model attacks and threats, security properties are nonetheless not explicitly

represented. Moreover, for threat analysis, attacks should be explicitly expressed and manually solved. [179] defines a formal basic set of security services for accomplishing security goals. In this approach, security properties analysis strongly relies on designer's experience. Moreover, threat assessment is not easily feasible.
In more recent efforts, temporal logic languages are used for expressing security properties. For instance, [75] embeds a first order Linear Temporal Logic (LTL) in the theorem prover Isabelle/HOL, making it possible to model both a system and its security properties. Although this is a rigorous approach, security properties and goals can be built upon security concepts, unfortunately leading to non-easily reusable specific models. Another example of temporal logic based verification is presented in [71]. In that approach the designer has to establish assumptions, knowledge and communication axioms, and represent them in a temporal logic language, and so specialized skills are definitely necessary. Additionally, the proposal only targets authenticity and is protocol oriented.
Evaluating both security and performance is tackled in [9]. If the methodology aims to offer a good trade-off between Quality of Service (QoS) and Security, it nonetheless requires a qualitative evaluation of security leaks. Indeed, when security flaws are detected, the designer must decide to use a new security mechanism - and so to degrade performance - or to keep the leak, in case the designer assumes that it is of low importance in the system. As a consequence, this approach strongly relies on designer skills and experience.
Assessment of security in embedded systems mostly relies on formal approaches. However, [145] mixes formal and informal security properties. The authors argue that unified security approaches won't provide enough flexibility to cope with highly heterogeneous requirements of distributed scenarios. Thus, the framework allows a user to define his own security properties and create dependencies between them, making the model probably difficult to reuse. Furthermore, the overall verification process is not completely automated, again requiring specific skills. Analogously, the Software Architecture Modeling (SAM) framework [10] aims to bridge the gap between informal security requirements and their formal representation and verification. Indeed, SAM uses formal and informal security techniques to accomplish defined goals and mitigate flaws. Thus, liveness and deadlock-freedom properties can be verified on LTL models relying on the Symbolic Model Verifier (SMV). Even if SAM relies on a well established toolkit - SMV - and considers a threat model, the "security properties to proof" process is not yet automated.
UMLsec [108] defines how to integrate security design elements, and security properties in a UML methodology. However, design elements and security properties are mixed on the same diagrams, as well as functional and non functional requirements. Also, UMLsec is not formally defined nor it covers threat analysis (e.g., attack trees).

As a conclusion, all these solutions usually make a trade-off between rigorousness and simplicity. On the one hand, security in embedded systems obviously requires rigorousness in the formal verification process. On the other hand, the complexity and diversity of systems and requirements, along with time-to-market and software-engineering criteria advocate for user-oriented tools with automated and simplified verification. AVATAR positively answers that need: it is based on a popular and friendly language (SysML), it is supported by an open-source toolkit (TTool) which relies on a recognized security verification toolkit (ProVerif). AVATAR further supports threat analyses and code generation, therefore supporting all secure system development phases: analysis, design, formal proof and code generation. On the contrary, other tools such as ST-Tool [89] and STA [50] are dedicated to one given system development phase, and based on non-reusable models.

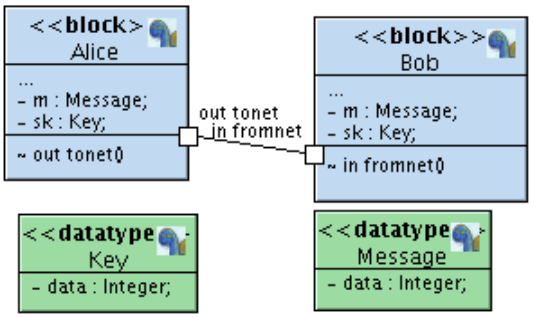


Figure 5.1: Example 1: Block diagram of the Alice and Bob system

5.4.2 Extending AVATAR for security purpose

Let's consider the following Alice and Bob system, in which Alice wants to send a confidential data to Bob, using a pre-shared symmetric key, and relying on an authentic message containing the ciphered confidential data of Alice. Figure 5.1 presents the SysML internal block diagram of this Alice and Bob system.

Several limitations make it difficult to prove confidentiality and authenticity properties in this toy system, and more generally in system design with security issues:

- **Initial knowledge**: It is not easy to model the pre-sharing of a key by Alice and Bob. Indeed, AVATAR provides no way to pre-share data, i.e., make data common to several blocks before the system starts. Cryptographic protocols often assume the pre-sharing of cryptographic data, e.g., cryptographic keys.
- **Cryptographic functions**: Cryptographic systems commonly rely on a set of well-known functions (symmetric cipher and decipher, compute MAC, etc.) that are not defined in AVATAR, and have thus to be explicitly modeled by a designer.
- **Communication architecture**: AVATAR channels cannot be listened by an external Block, i.e., a network on which an attacker could listen packets has to be explicitly modeled using blocks.
- **Attacker model**: AVATAR does not include any attacker model. Having a default attacker model, e.g., based on Dolev-Yao [73], would avoid users of AVATAR to have to model by hand an attacker model.
- **Security properties**: Security properties cannot be defined in AVATAR Requirement Diagrams, nor in TEPE, nor at Block Diagram level.

Previous limitations justify the following enhancements to AVATAR:

1. **Analysis stage**. Security requirements are captured within SysML Requirement Diagrams. Attacks are modeled in Parametric diagrams. This stage remains unformal.
2. **Design stage**. AVATAR blocks can be customized as "security blocks". The latter contains the definition of cryptographic methods. Global knowledge can also be modeled within pragmas listed in UML notes.
3. **Security property modeling**. Confidentiality and authenticity properties are formally defined within pragmas listed in UML notes. These properties are obtained by refining refined security requirements.

4. **Formal proof**. The AVATAR design enhanced with pragmas is translated into a ProVerif specification. Results given by ProVerif are represented directly in TTool.

All those stages are now more deeply explained. They are also further detailed in [156, 100, 158, 157].

5.4.2.1 Security analysis

SysML Requirement Diagrams have been extended to support security requirements [100] and parametric diagrams can be used to model attack trees. Additional benefits of this approach are a more precise definition of the use cases, the verification of existing attacks, the identification of new attacks, and a more explicit mapping of security requirements to functions and assets.
In our proposed framework, security requirements are modeled in SysML Requirement Diagrams (RD). Main operators of SysML RD are *Requirement Containment* and *Derive Dependency* formalisms used to define relationships between requirements. Also, the *containment* relationship can contain multiple sub-requirements in terms of hierarchy and enables a complex requirement to be decomposed into its containing child requirements whereas, *deriveReqt* determines the multiple derived requirements that support a source requirement. These requirements normally present the next level of requirement hierarchy. A *Security Requirement* stereotype is introduced to make clear distinction between functional requirements and security requirements of the system. In this way system engineers can model both functional and non-functional requirements of the system in one modeling environment. Furthermore, a *Kind* parameter is defined to specify the category of the security requirement such as, *confidentiality*, *access control*, *integrity*, *freshness*, etc,.
Attack trees can be modeled with slightly customized SysML Parametric Diagrams [152]. Attacks are modeled as values embedded into blocks representing the target of the attack. Attacks can be linked together with the following constraints: *or*, *and*, $after$, $before$, *sequence*. An attack can also be tagged as a *root* attack, meaning that this is attack is at the top of the tree. Last but not least, attacks can be linked to requirements, thus allowing an automated coverage of attacks.

5.4.2.2 Security design

- **Initial knowledge**

SysML offers several ways to share data between classes, using for example *block attributes*, or using a dedicated block storing that shared knowledge. Unfortunately, those solutions suffer two drawbacks:

1. The sharing is not really explicit, i.e., it is not clear which block intends to use a given block attribute, or given data of a dedicated block.

2. The sharing is defined for the entire system execution: in security, we are interested by the **pre** sharing of information, not by the sharing of this information during the entire system execution.

To overcome those two limitations, we propose to use specific directives - or pragmas - in notes of Block Diagrams. The pragma is as follows:

```
♯ InitialCommonKnowledge BlockID.attribute [BlockID.attribute]*
```

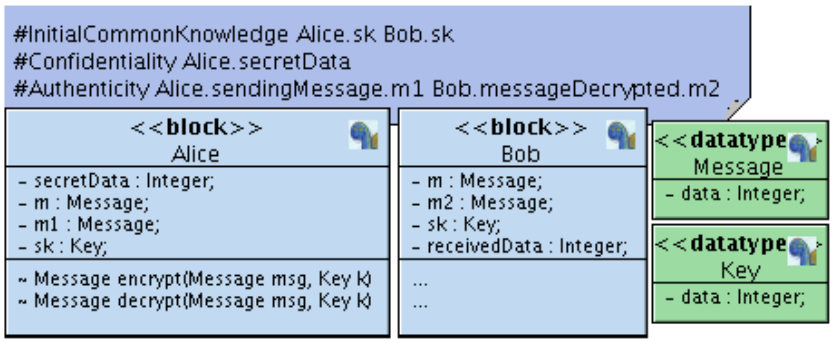


Figure 5.2: Example 2: Block diagram of the Alice and Bob system

Figure 5.2 contains the internal block diagram of the Alice and Bob system enhanced with AVATAR extensions for security. Similarly to the first design depicted in Figure 5.1, the new design has two blocks (*Alice* and *Bob*), and declares two data structures (*Key* and *Message*). A note now declares that *sk* is a pre-shared data (a key) between Alice and Bob:

```
# InitialCommonKnowledge Alice.sk Bob.sk
```

Sometimes, the knowledge is not common to all protocol sessions, but is initialized for each protocol session. The following pragma can be used to handle this situation:

```
# InitialSessionKnowledge Alice.sk Bob.sk
```

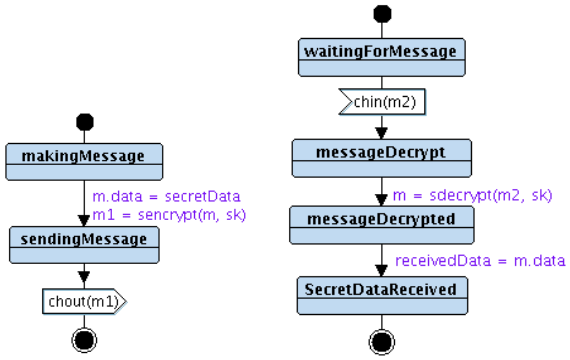
- **Cryptographic functions**

AVATAR includes the definition of a specific block called *cryptographic block*. That block defines a set of cryptographic functions that can be used as regular methods by the state machine of these blocks (some methods are visible in Figure 5.2). For example, both Alice and Bob declare a set of cryptographic methods. A few examples of cryptographic functions we have defined:

- $encrypt(Message\ msg, Key\ k)$ and $decrypt(Message\ msg, Key\ k)$, for encrypting and decrypting messages with asymmetric keys, respectively.
- $sencrypt(Message\ msg, Key\ k)$ and $sdecrypt(Message\ msg, Key\ k)$, for encrypting and decrypting messages with symmetric keys, respectively.
- $MAC(Message\ msg, Key\ k)$ and $verifyMAC(Message\ msg, Key\ k, Message\ macm)$, for computing the MAC of a message, and verifying the MAC of a message, respectively.

For instance, the behavior of *Alice* and *Bob* is provided within two respective State Machine Diagrams (see Figures 5.3-a and 5.3-b, respectively). Alice first puts its *secretData* into a message $m.data = secretData$, then encrypts this message $m1 = sencrypt(m, sk)$ with the symmetric encryption function, and finally sends the resulting message on the broadcast channel $chout(m1)$. *Bob* waits for a message on the broadcast channel $chin(m2)$. Then, *Bob* tries to decrypt the received message $m = sdecrypt(m2, sk)$ and then extracts from the message the *secretData* sent by Alice: $receivedData = m.data$.

- **Communication architecture**



a – Alice b – Bob

Figure 5.3: Example 2: State Machine Diagrams of the Alice and Bob system

AVATAR communications are based upon unidirectional one-to-one synchronous or asynchronous communications between blocks. Security extensions consists in adding the possibility to tag a link with *private* or *public*. A public link can be listened at by an attacker, a private link cannot. If a public link is declared as connecting a block to itself, then all sub blocks of that block can use that link: it can be used to model a network common to a set of entities.

- **Attacker model**

In AVATAR, the attacker model is implicit, i.e., there is no need to model an attacker either at Block Diagram level, or at State Machine Diagram level. Indeed, AVATAR relies on the attacker model of ProVerif, see section 5.4.2.4.

5.4.2.3 Security properties

TEPE [123] has already been proposed for modeling safety-related properties in AVATAR. However, safety properties are commonly complex to express. Indeed, they generally involve several attributes and signals of blocks, as explained in [123] on an elevator system. On the contrary, security properties can be usually defined with a type (e.g., confidentiality), and with elements related to that kind (e.g., the confidentiality of the attribute of a block). This property simplicity advocates for a basic modeling solution, that is not based on complex diagrams or operators. Finally, our solution relies on *pragmas* provided in notes of Block Diagrams: *confidentiality* and *authenticity* can be directly expressed at this level. These low level properties are expected to be obtained after a refinement of high-level security requirements, during an iterative process.

- **Confidentiality**

  Confidentiality in AVATAR can be modeled as a simple pragma provided in the note of a Block Diagram. Confidentiality must be specified as the confidentiality of an attribute of a block:

  ```
  ♯ Confidentiality block.attribute
  ```

  Coming back to the example provided in Figure 5.2, the following statement models that the attribute *secretData* of *Alice* shall remain confidential i.e., never accessible to an attacker:

```
♯ Confidentiality Alice.secretData
```

- **Authenticity**

  Authenticity in AVATAR is also modeled as a pragma. An authenticity pragma states that a message $m2$ received by a block $block2$ was necessarily sent before in a message $m1$ by a block $block1$. The authenticity pragma specifies two states: one of the sender block, i.e. one state $s1$ of $block1$, and one state $s2$ of $block2$. Also, in the state machine diagram of $block1$, $s1$ corresponds to the state **right before** the sending of $m1$. Analogously, $s2$ corresponds to the state **right after** message $m2$ has been received and accepted as authentic. Finally, the authenticity pragma is as follows:

  ```
  ♯ Authenticity block1.s1.m1 block2.s2.m2
  ```

  For example, in Figure 5.2, the authenticity pragma states that all messages $m1$ sent by *Alice* after state $sendingMessage$ shall be authentic for *Bob* receiving it into a message named $m2$ before its state $messageDecrypted$.

  ```
  ♯ Authenticity Alice.sendingMessage.m1 Bob.messageDecrypted.m2
  ```

5.4.2.4 Formal proof of security properties

Several environments target the proof of security properties, e.g., SHVT [149], AVISPA [37], and ProVerif [46]. Proofs within the SHVT environment cannot be automatically conducted: so, it was excluded. In AVISPA, the tool is system dependent, and really focused on cryptographic protocols, which is not the case of AVATAR which targets embedded systems in general. ProVerif is based on process algebra, and is therefore well suited for modeling communicating entities as found in embedded systems. Also, to our experience, ProVerif nicely solves the trade-off between expressiveness, complexity and automation of formal approaches. We finally selected ProVerif, but AVISPA may also have been used.
ProVerif [46] is a toolkit that relies on Horn clauses resolution for the automated analysis of security properties over cryptographic protocols. ProVerif takes as input a set of Horn Clauses, or a specification in pi-calculus (a process algebra) and a set of queries. ProVerif outputs whether each query is satisfied or not. In the latter case, ProVerif tries to identify a trace explaining how it came to the conclusion that a query is not satisfied.
A ProVerif specification contains *pi-calculus processes* and properties represented as *queries*. Our approach takes as input an AVATAR design augmented with design pragmas and property pragmas, and outputs a ProVerif Specification. The whole process is seamlessly implemented in TTool.

- **Processes** are composed of a *declaration part*, of *a definition of a set of sub-processes*, and the definition of a *main process*.
  The declaration part can be used to declare global terms, including channels, and functions.
  Main process operators are summarized in Table 5.1.

- **Properties** are represented with ProVerif *queries*. Queries are formal clauses in which the left hand side of the implication is a set of facts that should be accomplished whilst the right hand side includes the hypotheses to be verified. Queries can be used

Table 5.1: Main ProVerif process operators

Construct	Semantics
`out(c,M); P`	Sending of M in channel c, and execution of process P, i.e., $\bar{c}\langle M\rangle.P$.
`in(c, M);P`	Receiving of Message M from channel c, and execution of process P, i.e., $c(M).P$.
`new a; P`	Definition of a new term a, and subsequent execution of process P, i.e., $(\nu a)P$.
`begin(M); P`	Execution of an event, and subsequent execution of process P, i.e., $begin(M).P$.
`!P`	Replication of process P, i.e., an infinite number of instances of P are executed.
`P|Q`	Parallel composition between processes P and Q.
...	... other pi-calculus operators, such as *if then* statements, etc.

to express confidentiality [45] and authenticity requirements [46].
Confidentiality queries directly express which data shall not be accessible to the attacker, e.g., that a private key shall not be accessible to an attacker:

```
query attacker:myKey.
```

Authenticity of messages relies on ProVerif events. Whenever a message `m` sent by a process `A` to a process `B` shall be authenticated, one event shall be included in each process: one shall be included in `A` before the sending of `m` (e.g., `eventSendM`), and one after the receiving of `m` (e.g., `eventReceiveM`). Since the attacker is not allowed to execute events, it suffices to prove that to each receiving event of `m` corresponds exactly one sending of `m`. And so, an injective query is used to model authenticity properties:

```
query evinj:eventReceiveM(x) ==> evinj:eventSendM(x).
```

Important note: ProVerif also makes it possible to study the reachability of events, based on queries.

ProVerif integrates its own **attacker model**, which is itself a pi-calculus process implementing a Dolev-Yao approach [73]. This process acts like an adversary relying upon a set of known names, variables and terms, i.e., the attacker *knowledge*. To increase its knowledge, an attacker relies on public channel probing and execution of functions non prohibited to the attacker. And so, ProVerif is not intended to study computational attacks: CryptoVerif [48] could be used for that purpose.

To verify a query, ProVerif implements a resolution algorithm [46], [3] [47] that first translates the complete pi-process specification to Horn Clauses [46]. To verify a query, the resolution algorithm determines, based upon a set of inference rules, if the attacker reasoning is able to derive a trace that contradicts the query, thus proving that the query is false. Otherwise, if the attacker is unable to find such a trace, then the property is satisfied. Additionally, if facts on which the query is based upon are not reachable, the

algorithm informs that the query can not be proved.

5.4.3 Basics of translation:

Again, the translation process takes as parameter an AVATAR design, including its sets of pragmas, and outputs a ProVerif specification.

Briefly, the translation process is as follows: Let $\mathcal{T}$ the translation process that takes as input a Block Diagram BD, and a set of pragmas P, and Pr the resulting ProVerif specification:
$Pr = \mathcal{T}(BD, P)$.

- A BD is composed by three graphical entities named $<<$ *block* $>>$, $<<$ *datatype* $>>$ and $<<$ *pragmas* $>>$. A block contains a set of attributes, a set of functions, a set of signals and a reference to a State Machine Diagram (SMD). $<<$ *datatype* $>>$ can be ignored for the translation process since they can easily be removed.
- An SMD is a set of interconnected logical operators: *start states*, *stop states*, *transitions* - with attribute settings and function calls -, *choices*, *states*, *sending in a channel*, *receiving from a channel*.
- The type of a pragma in P is either $InitialCommonKnowledge$, $InitialSessionKnowledge$, $Confidentiality$, or $Authenticity$.

$\mathcal{T}$ applies the following set of rules:

1. For each block $b \in BD$, a "first" process `fp` is generated. Then, for each state s of the State Machine Diagram *smd* of b, another process `ps` is generated.
2. `fp` instantiates all attributes that are not listed in $InitialCommonKnowledge$, $InitialSessionKnowledge$ or $Confidentiality$ pragmas: '`new attr;`'. Then, `fp` makes a call to the `ps` process corresponding to the start state of *smd*.
3. Each `ps` is created as follows. An event is first called for tracing the reachability of states '`event entering_state_nameofs();`'. Then, each branch of logical operators linked from s is taken into account until another state is reached on that branch:
   - Sending on a channel c of a message m is translated as an '`out(c, m);`'.
   - Receiving on a channel c of a message m is translated as an '`in(c, m);`'.
   - The assignment of a variable is translated using a '`let`' operator, e.g.: '`let m1.data = m2.data;`'.
   - The call of a cryptographic function is translated with a ProVerif cryptographic function and a '`let`' operator: '`let mac = MAC(msg1.data, Key.data);`'.
   - The call of a non-cryptographic function is translated with a simple call to an event having the name of the corresponding function, and with the same parameters, e.g., '`event function(par0, par1);`'.
   - The various branches starting from state s are translated based on '`if...else`' ProVerif statements.

4. The main process `mp` of the ProVerif specification instantiates all attributes listed in *InitialCommonKnowledge* pragmas. Then, it instantiates in parallel, and for an infinite number of sessions, all `fp` processes, e.g., '`(!fp`$_1$`)|(!fp`$_2$`)|...|(!fp`$_n$`)`'. In case an *InitialSessionKnowledge* pragma is used for attrributes, then the instanciation of these parameters is done in each session: '`! (new att; (fp`$_1$`)|(fp`$_2$`)|...|(fp`$_n$`))`'

5. *Confidentiality* pragmas referencing a block b and an attribute *attr* of b are translated as a declaration of *attr* as follows: '`private free attr.`' and with a query of the following form:
'`query attacker:attr.`'

6. *Authenticity* pragmas of the form $b1.state1.attr1 b2.state2.attr2$ are translated using statements of the following form:
'`query evinj:b2_state2(attr2) ==> evinj:b1_state1(attr1).`'.
Additionally, in the process `ps` where $s = s1$, a call to '`event b1_state1(attr1);`' is added at the beginning of the process. Similarly, a call to '`event b2_state2(attr2);`' is added at the beginning of the process `ps` where $s = s2$.

5.4.4 Results

Our contributions on the proof of both safety and security properties from the same models have been recently published, and so, only a few case studies have been successfully performed. For example, we can underline that several quite complex cryptographic protocols defined for secured automotive embedded architectures have been modeled and proved against security properties using TTool/AVATAR [158, 157, 82]. An overall MDE-based methodology for the development of secured embedded systems is still an open issue that we intend to tackle in the incoming years.

5.5 Prototyping

5.5.1 Context and problematic

The prototyping of an embedded system is usually a cumbersome task, especially when multiple hardware targets must be taken into account. The first step is usually to generate an executable code for the local platform, and to execute it. Unfortunately, strong differences between the local and the target platforms may reduce the effectiveness of that step. Another approach is to rely on a virtual prototyping platform which can execute in a more realistic way the application.
Prototyping platforms have been proposed at different levels of abstractions.
At a very high-level of abstraction, the DIPLODOCUS/TTool approach [33] targets the design space exploration of System-on-chip. Application functions can be mapped on abstract CPUs or hardware accelerators, and then can be evaluated with simulation [114] or formal verification techniques [121, 122]. However, results that can be expected at that level of abstractions are related to bus or CPU loads, rather than to a precise timing execution on a hardware platform.
SystemC is a widely spread set of C++ libraries and simulation kernel for modeling and implementing electronic systems [91] [144]. Several levels of abstraction have be defined in SystemC, ranging from transactional level modeling to a cycle accurate level modeling. The SoCLib library of component models [175] is based on SystemC. SoCLib supports two

levels of abstractions: TLM (Transaction Level Modeling) and CABA (Cycle Accurate Bit Accurate). Other open prototyping platforms like SkyEye [173] and QEmu are not based on SystemC components.

5.5.2 Our approach: rationale and overview

Our prototyping approach relies on high-level models (i.e., AVATAR models), on code generation, on an embedded operating system (MutekH), as well as on a SoC prototyping platform (SoCLib). The overall approach, and a few results, have been published in [16, 17] and are available in the latest release of TTool.
More precisely, our AVATAR-prototying approach is based on:

- A C/POSIX code generator. TTool can generate C/POSIX code from SysML models, and link it against libraries implementing AVATAR features (synchronous communications, timers, etc.).

- An open platform for the virtual prototyping of complex Systems-on-Chip: SoCLib [175]. SoCLib is a SystemC library of component models. It supports several models of processors (Mips, Arm, PowerPC, Sparc, MicroBlaze, etc.), of buses, of memories, and several operating systems, including eCos, MutekH and RTEMS. MutekH is an embedded operating system used on multiprocessor platforms in various research projects. It was originally designed with native support for processors heterogeneity in mind [135]. SoCLib supports two simulation models (Transaction Level Modeling and Cycle Accurate Bit Accurate), and comes with debugging features like a GNU debugger server and a memory access checker similar to Valgrind.
  Multiple operating system projects are available for use as a target POSIX platform for code generation which have different features and memory footprints. As our approach is based on free software projects for modeling or performing simulations and proofs, the operating system running on the target platform had to be free. This thus rules out VxWorks and similar proprietary operating systems from the scope of our demonstration even if the use of such software is technically possible in our toolchain. The SoCLib prototyping platform has a number of supported operating systems, including UNIX like implementations such as NetBSD as well as some lightweight embedded operating systems.

  NetBSD and various Linux flavors are system-call based and require a separate set of user library packages in order to build and run applications. System start-up of such large operating system kernels running on top of a SystemC simulator may require a large amount of time.

  Most embedded operating systems can be used without system-call interfaces, the application is thus compiled along with the kernel and all objects files are linked in a monolithic binary file. Some well known operating systems based on this approach include FreeRTOS, eCos and RTEMS. eCos and RTEMS may suit our needs because of their support in SoCLib, and their implementation of the thread POSIX interface. We expect to support them in the future. The operating system chosen in the scope of AVATAR is MutekH. It offers similar features: it's highly configurable and comes with suitable libraries, and with the additional benefit of heterogeneous multiprocessors support and a better SoCLib integration.

  Moreover, when used along with the SoCLib platform, MutekH can be configured to provide necessary information related to memory allocation and execution stack

boundaries to the *MemoryChecker* SoCLib module. This information along with details of memory accesses performed by the processors allow this module to track suspicious memory access and report them to the developer. This feature is of great help on hardware were no Memory Management Units can be used which prevents relying on memory protection to find bugs or undersized stacks [161]. Those debugging features apply to both operating system and user codes. In our approach, the user code refers both to the generated code and also to the C code directly provided by the user as design model parameters.

From an Avatar design model, TTool generates a C/POSIX code at the push of a button. The code can be compiled for the local host, or for the SoCLib platform: in that latter case, MutekH, the generated code and the AVATAR runtime are compiled and linked against the generated C/POSIX code.
The prototyping phase is intended to be applied on refined models. Indeed, the prototyping phase is particularly useful to evaluate whether a given hardware platform is well suited to execute a given set of software components.
A refined model is a model in which some abstractions of a more abstract model have been resolved. For example, AVATAR designs make it possible to abstract algorithms with their estimated durations: a *computeFor(minDuration, maxDuration)* can be added to state machines transitions. Another example of abstraction is to let branches of choices undetermined, that is, at a high level of abstraction, all branches of choices may be considered. At formal verification level, this means that all branches have to be explored. But on a more refined model, branches of a choice are not randomly taken, but they are usually rather selected according to the result of operations. Finally, abstractions shall be resolved before doing the prototyping phase. To do so, an AVATAR user could use the AVATAR state machines to put more information in its model. Unfortunately, when coming to complex algorithms - e.g., in our case, cryptographic algorithms - , a graphical model based on state machines is not practical. Therefore, the best option is probably to directly replace given elements of an AVATAR design with its corresponding implementation code, e.g. replacing a *computeFor(minDuration, maxDuration)* by the C algorithm it models. If this C code included into the model is actually ignored by the integrated simulation and formal verification capabilities of TTool, this code can be automatically included in the C/POSIX code generated by TTool.
Finally, the most abstract AVATAR models performed with TTool generally represent the control part of applications, and can thus be simulated and formally verified. On the contrary, more refined models resolve non determinism behaviors with low-level representations (e.g. in C) of data and algorithms. Those refined models are difficult - if not impossible - to simulate and to formally verify, but are definitely useful for prototyping purpose.

5.5.3 Code generation

Basically, the C/POSIX code generator of TTool works as follows (More details are given in [17]):

- One *.c* and one *.h* file which contains a representation of the state machine are generated for each block. The translation of operations on variables, method calls and tests is quite straightforward. On the contrary, synchronous data exchange,

asynchronous data exchange, and time manipulation are more complex and are thus handled by the AVATAR library (i.e., the AVATAR runtime).

- The main file (*main.c*) is in charge of defining one thread per block, setting the attributes of those threads (e.g., on which CPU each thread must be executed, which scheduling policy to use, etc.), starting all threads, and finally waiting for their termination.

5.5.4 The AVATAR runtime

The AVATAR runtime is a set of libraries that handle all synchronous and asynchronous communications between blocks. Basically, it relies on data structures to store requests from blocks, and on mutex and condition variables to achieve necessary synchronization between threads of blocks. Its implementation is lightweight (about 2000 lines of C code). The AVATAR runtime is automatically linked against the generated code when compiling the latter.

5.5.5 Example with an automotive application

The AVATAR prototyping is illustrated with an automotive embedded system designed in the scope of the European EVITA project [1]. Recent on-board Intelligent Transport (IT) architectures comprise a very heterogeneous landscape of communication network technologies (e.g., LIN, CAN, MOST, and FlexRay) that interconnect in-car Electronic Control Units (ECUs) [171] [169]. The increasing number of such equipments triggers the development of novel applications that are commonly spread among several ECUs to fulfill their goals.
An automatic braking application serves as a case study [109]. The system works basically as follows: An obstacle is detected by another automotive system which broadcasts that information to neighbor cars. A car receiving such an information has to decide whether it is concerned with this obstacle, or not. This verification includes a plausibility check function that takes into account various parameters, such as the direction and speed of the car, and also information previously received from neighbor cars. Once the decision to brake has been taken, the braking order is forwarded to ECUs responsible for performing the emergency braking. Also, the presence of this obstacle is forwarded to other neighbor cars in case they have not yet received that information. The model of the active braking system can be prototyped as described here. The generated source code is usually first prototyped on the local platform. A simple reason for this is that the final hardware target may not yet be available, or even clearly defined. However, when the hardware platform has been specified, the prototyping phase is as follows:

1. **Generation of the cross-compiler**. A cross-compiler for the target platform must be generated. In our example, we have used *gcc*-based cross-compilers. In the scope of our example, we have prototyped the active baking application on various 32-bit processor architectures, including PowerPC, Arm, Mips and Sparc.

2. **Generation of the C/POSIX code**. From an AVATAR model in which non deterministic behaviors have been resolved (ideally), TTool generates a set of *.c* and *.h* files, as explained in previous sections. The main file describes how threads are mapped on the different CPUs.

3. **Compilation of the code**. The generated C code, the AVATAR runtime, and MutekH are compiled with the cross-compiler, and linked together as one executable file. The executable file could obviously be run on the real hardware or a virtual prototyping platform built using SoCLib.

4. **Prototyping with SoCLib**. The SoCLib simulator is started with the desired hardware configuration which runs the executable file generated at previous step. Our example has been tested on several processors: PowerPC, which are wildly used in automotive systems, but also Mips, Arm and Sparc processors. In all cases, we have used a 5processor configuration: one CPU per ECU, and one CPU to execute the environment blocks.

5. **Result analysis**. Results of the prototyping simulation can be visualized either in the console, or directly in TTool as a UML sequence diagram (see Figure 5.4). The GNU debugger *gdb* can also be used to have more information about the execution of the code, e.g., about memory allocations, to perform step-by-step execution, to monitor which threads are currently executing, etc. Using traces, important prototyping information can be obtained. In our case, the latency between the receiving of an emergency message and the corresponding braking action can be clearly evaluated for each processor type.

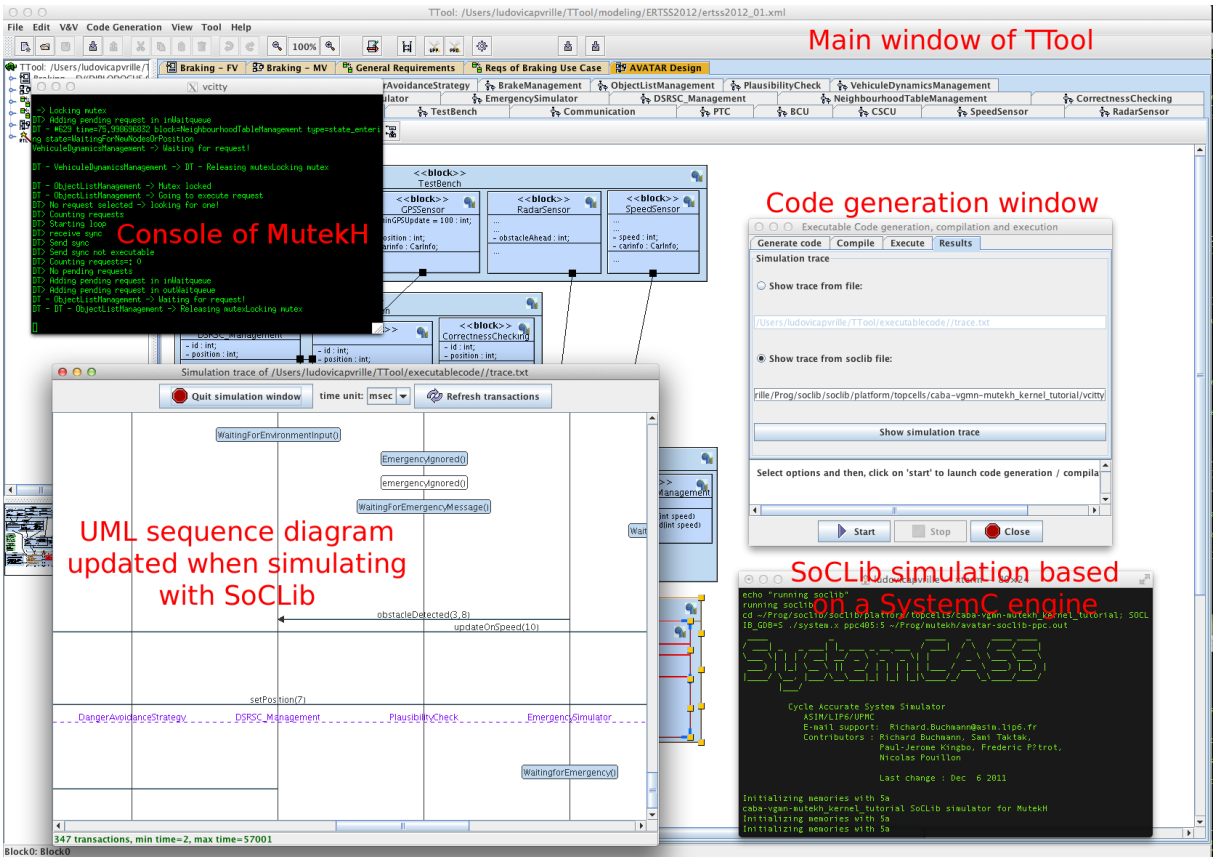


Figure 5.4: Prototyping environment based on TTool, MutekH, and SoCLib

5.6 Conclusion

The combination of AVATAR/SysML/TTool, UPPAAL, ProVerif and SoCLib/MutekH offers an integrated platform for embededd systems engineering. Indeed, this platform offers at the same time a well-known modeling language (UML, SysML), an easy-to-use proof environment and a prototyping simulation platform comprising several commonly used microprocessors and operating systems. Simulation, proofs of both safety and security properties, and prototyping can be performed at the push of a button and their results are displayed directly in the model. This research work has been published in several conferences [156, 100, 158, 157, 16, 17] and used in the scope of the EVITA project [83, 82]. Most TURTLE users have now switched to AVATAR, and return from experience is really encouraging. Main remark is about a better integration of safety and security. Indeed, security proof is still limited to a few properties, and some AVATAR operators must be avoided to allow ProVerif to more easily come to an answer about security properties. Work will address those issues in the incoming years.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

6.1.1 Discussion

So many (too many?) contributions have addressed the modeling and verification of complex systems. Many formal languages were introduced in the 80's, using a wide variety of formats. Each of them was commonly adapted to only one kind of systems (e.g., *Estelle* for communication protocols), or was targeting one given modeling aspect (e.g., state charts for behaviour modeling). Progressively, modeling notations were enhanced with graphical elements (operators, diagrams, views), with models of computation and communication, and with methods. Some of these graphical notations finally merged in the mid 90's to create the Unified Modeling Language. The definition of the latter being informal, and not adapted to many application domains, several domain-specific profiles were subsequently defined - Some of these profiles formally. Methodologies on how to use such profiles were also widely published. TURTLE was part of this past adventure.
Since my Ph.D. defense in June 2002, the OMG has released several well-known language specifications (e.g., SPT, SysML, MARTE) with the same limitations (e.g., lack of methodology, lack of formal definition). Model-Driven Engineering is probably the main contribution of the last decade in modeling approaches. MDE targets system analysis, modeling, simulation, code generation, and documentation. MDE relies on the UML language, and on meta-modeling in order to define Domain-Specific Languages. Model Driven Architecture specifically targets two abstraction levels: Platform-Independent Model (PIM) and Platform Specific Model (PSM). PSM is constructed from PIM with model transformation techniques, and executable code is expected to be generated from PSM. Raising the abstraction level (PIM) combined with model transformation techniques help simplifying portability, interoperability, documentation, and maintenance. Portability is a direct consequence of a higher abstraction level. Interoperability is due to the independence of PIM from any hardware or middleware support, thus authorizing PSM components created from PIM-level models to exchange information. Documentation can be generated from both PIM and PSM levels. Documentation includes argumentation for choices: the higher the abstraction level, the most important is the documentation on choices, because late re-engineering is more costly. Maintenance can be done on both abstraction levels. But contrary to documentation handling, in case of identified problem in the software, maintenance shall first be performed on the PSM, and if necessary in the PIM. Indeed, the higher in abstraction levels modifications have to be performed, the most costly they are.

Extreme programming (XP) [41] is another software-centric methodology. XP is based on a defensive programming approach, with small step-by-step code increments strongly tested and documented. Documentation is given within the code. This approach partially avoids dealing with models, and thus it avoids working at different abstraction levels in which model coherency and refinement may slow down the software development task. XP also allows for formal proof with static code analysis techniques. However, even in extreme programming, testing and verifications must be done according to requirements that need to be captured. They are commonly captured either in databases or within models: in XP, annotations can be used to reference requirements. But tracability and refinement of requirements is something that extreme programming can definitely not easily address. Nor the allocation of resources which has to be evaluated along the code creation. XP tries to address some these issues mentioned above using annotations that are semantically close to concepts found in the MDA approach. Yet, the XP approach can be seen as a much lighter version than more complex MDE-based methodologies (e.g., Rational Unified Process). Finally, mixing code and annotations mixes elements with strong semantical differences, which explains that we do not support that kinds of methodology for critical systems.
Another software-based methodological approach is called *Agile Software Development* [189]. Software ability to handle specification changes is one of the main focus of Agile. Agile includes an iterative and incremental process until the final code release. Exchange between developers is expected to frequently occur. Models are not really mentioned in Agile, apart from documentation. Indeed, models cannot really be executed and tested like software, and usually require extra work w.r.t. direct coding. A few initiatives target the reconciliation between Agile and modeling (e.g., Agile Modeling, and Agile MDA).
Finally, new software-centric graphical engineering techniques have been introduced during the last decade, and in particular the MDE/MDA approaches standardized at OMG. Our contributions do not rely on meta-modeling, but rather on the profiling of UML. We have taken that research path because meta-modeling has not yet - in our opinion - demonstrated that software-development teams are ready to develop their own modeling environments, for several reasons we will discuss in the future work section. However, concepts behind MDE, in particular the different levels of abstraction of MDA, have been taken into account in our research work. Also, our contribution does take hardware constraints into account. Last but not least, our contributions are settled upon the most recent OMG modeling languages: SysML and MARTE, in particular.

6.1.2 Recall of contributions

This document contains an overview of research work I have conducted since my Ph. D. defense, a decade ago, in the field of Model-Driven Engineering and of Model Driven Architecture. My Ph.D. thesis sketched a first semi-formal UML profile for the modeling and formal verification of time critical embedded systems. Since then, a lot of work has been achieved, always with the same main objective: simplifying proofs along a system engineering processes targeting the development of complex embedded systems.
First, to fully support an engineering process, from system dimensioning until system deployment and code generation (chapter 2). The engineering process is now supported with an open-source software (TTool), and has been experimented in several academic, publicly funded research projects and industrial partnerships.
Second, a new approach has been developed for the design space exploration of Systems-

on-Chip: DIPLODOCUS (chapter 3). DIPLODOCUS is based on two abstraction levels that facilitate the modeling of Systems-on-Chip: Application level (PIM), Architecture and Mapping levels (PSM). DIPLODOCUS follows several concepts introduced in the MARTE OMG profile. DIPLODOCUS supports very fast simulation and formal verification techniques. Also, we have developed our own formal verification toolkit that supports variable model coverage. DIPLODOCUS has been used in the scope of several projects, and has received several industrial grants. TTool fully implements DIPLODOCUS, including its coverage-enhanced simulator.
Third, although much work has been achieved on system modeling, requirements and property capture in high-level languages has not been tackled efficiently up to now. Requirement capture is still not covered enough in MDE processes, which probably explains contributions proposed outside of MDE processes, e.g., KAOS. Our research work tackles the efficient capture of functional and non functional requirements within SysML Requirement Diagrams. It also addresses graphical property modeling based on Timing diagrams or Parametric Diagrams. The automatic generation of observers or CTL formula from property models has been experimented in both TURTLE and DIPLODOCUS (chapter 4). Two Ph.D. thesis have focused on this issue [79, 120], and new results are expected in the incoming year, in particular, in the security field [100].
Last but not least, security issues is now handled within a new profile called AVATAR (chapter 5). Security threats are now clearly at stake in many critical embedded systems. Yet, safety and security issues are rarely addressed in the same modeling environments. AVATAR tackles offers the possibility to perform both safety and security proofs from the same system models. AVATAR has been succesfully used in the scope of automotive systems, and is now used worldwide for teaching activities in engineering schools and universities, and trainings to engineers.

6.2 Future work

6.2.1 Discussion

Most contributions around MDE commonly address the same old problem, that is, offering the graal methodologies and modeling environments for addressing the safety issues in complex, distributed and real-time embedded systems. Timing constraints analysis, scheduling analysis, resource allocation, and concurrency analysis are commonly handled by these environments and methodologies. MDE targets application domains such as information systems, embedded systems, web-based systems, and graphical user interfaces. However, for companies developing systems in those domains, the interest for them to use meta-modeling techniques, or specific profiles for software-based systems, is not yet obvious, and following questions still need to be answered.

- Is it really worth training engineers to these new techniques, i.e. what kind of gains with MDE can really be expected with regards to currently used software development methodologies? For what kind of projects is it really worth? Can modeling skills be reused between projects? Can MDE really be an answer to handle software integration when (meta-)models or profiles have not been made by teams working in the same departments or companies? From a purely financial perspective, it is still to be demonstrated that MDE increases the margin of companies. Moreover, it already has been demonstrated that maintenance is probably one of the most costly part of

software development ... and this is probably also the development phase which still needs to be addressed in a more thorough and precise way by MDE.

- If a company decides to use modeling techniques, what is the real interest in using MDE techniques, and not simply use UML models as cartoons, i.e., as documentation mainly, for showing things rather than writing them. Of course, keeping away from MDE forces to use good old specific techniques for software parts that require simulations, and more rarely formal verification. We guess a high percentage of models performed in companies are used for documentation purpose only, and with regard to that, one important challenge companies face is to handle the coherency between models (or documentation) and what is effectively in the released application code.
- Tool useability and maturity is another challenge that must be addressed. Free meta-modeling tools are already available, and exchange of models and meta-models between tools is something that has progressed during the last years. However, meta-modeling is only used to define the semantics of operators and diagrams, and definitively not to define methodologies that are expected to be used along with the defined operators. And so, research work shall now address the adaptation of software-development processes according to in-house meta-models. Coherency between the meta-models and processes, and processes interoperatibility are other open issues.

Our contributions intend to offer a good compromise between a too specific approach limited to one class of systems or properties, and a too open one based on meta-modeling, that requires more advanced skilled to be efficiently used. We also expect our simple but formal profiles to be easily adapted to well-known software-centric methodologies. Yet, our contribution, with the handling of PIM and PSM in the DIPLODOCUS profile, and the definition and support of software-centric methodologies follows the MDE approach supported by OMG.

However, since a lot of research work has been achieved on the design part of systems development, contributions now mostly focus on how to integrate other system modeling issues around a system-design oriented ecosystem: property modeling, robustness, security, for example. Thematics listed in last national and european project calls on system modeling clearly reflect that trend. For example, the last national call on future aeronautics platforms (CORAC) clearly focused on how to integrate security and certification issues into a well-known safety-oriented design methodology.

This sketches out our future work for the incoming years.

6.2.2 So, what's next in a short term?

More concretely, our research work is oriented according to contract and project opportunities in the field of semi-formal environments for integrated and embedded systems. This research field is still challenging since it is a common belief that raw formal methods are scarcely used in industrial contexts. Thus, we still think formally-based model driven engineering is still to be investigated for enhancing its industrial acceptance.
More precisely, in the scope of my research work, I foresee three topics of interest for the next few years.

6.2.2.1 Static model analysis

Most contributions on semi-formal environments, including ours, are based on model transformation techniques, and then in the execution of the transformed model so as to verify properties. The raw use of formal methods often lead to combinatory explosion. Unfortunately, the use of automatic model transformation does not resolve that issue since the transformation process commonly ignores the combinatory explosion issue.
A first solution we have explored relies on an efficient combination of formal verification techniques. For example, whenever a modeling pattern that is known for favourizing combinatory explosion in a given verification technique is detected at transformation stage, then, another verification technique is used. We have been working on that issue for a few years. Our first idea was to combine model-checking techniques with first order logics [18]. In particular, when transforming a UML model in LOTOS, it is common to have recursive call of processes: this scheme is not supported by CADP: This particular situation was specifically targeted with [18].
Another technique we have explored relies on well-know static analysis techniques, that we have transposed to UML models. For example, in the scope of DIPLODOCUS, the coverage-enhance simulator uses a set of static model analysis techniques before executing the model: impact of variables (live variables), etc. We do think that some others static analysis techniques could be used, such as the **invariant technique** defined years ago for Petri Nets. We have already started to work on that issue, and we expect new results in a few years.

6.2.2.2 Security properties

- The identification of attacks and requirements is of utmost importance to build secure embedded systems. First proposals have been made so as to efficiently capture attacks and requirements based on a first high level architecture design [100]. Yet, an efficient traceability and refinement of attacks and security requirements in model-driven engineering is still an open issue that we intend to tackle in the incoming years.

- AVATAR is a UML profile dedicated to the proof of both safety and security properties. Currently, only confidentiality and authenticity properties are supported. When working on the EVITA projects, many requirements could be refined in confidentiality or authenticity properties. But some others could not, in particular, requirements dealing with *freshness* or *integrity*. Research work must therefore be conducted in formal description techniques handling those properties, and on model transformation so as to extract relevant information from AVATAR models. Other kinds of systems could benefit from this work: information systems in particular.

6.2.2.3 Energy consumption

If the DIPLODOCUS approach can efficiently take into account performance constraints (e.g., "can a chip decode a HD video?"), power consumption issues are insufficiently addressed. Now that those issues are at stake in embedded devices - *greenIT* -, we intend to integrate power consumption criteria in DIPLODOCUS. In particular

1. Software (*power managers*) and hardware (*power controllers*) mechanisms dedicated to power management shall be analyzed in several hardware platforms for mobile devices. The result of this study shall be a proposal of how to efficiently abstract

those power management techniques, so as to invent new UML diagrams - using meta-models - dedicated to model power managers and controllers. The relation between those diagrams and the ones defined in MARTE [154] shall also be studied.

2. The definition of techniques for performing formal proofs from models defined at previous step. Formal proofs shall not consider any hardware / software implementation at that stage, but only functions performed by power managers and controllers. In other words, the idea is to prove (safety) properties on a power management system independently from its implementation.

3. The integration of power managers and controllers in DIPLODOCUS abstract architectures. Once again, the idea is to define meta-models for positioning power management functions in a software / hardware embedded architecture : interaction with operating systems running on CPUs, interaction with hardware components i.e., how the power controller drives execution modes of CPUs, memories, etc.

4. The definition of simulation and formal proof techniques for validating the previous stage, i.e., the integration of power management functions into a hardware / software architecture. In particular, the objective is to prove that a given power management integration does not lead the system to a wrong functioning mode (e.g., deadlock situation).

Thus, We expect to enhance TTool / DIPLODOCUS with energy-aware diagrams, and in particular it shall be possible to capture under TTool - using UML diagrams - the characteristics of power managers and controllers, and to verify them with a press-button approach.

Bibliography

[1] The EVITA european project. http://www.evita-project.org/.

[2] D. Akhvlediani A. Pop and P. Fritzson. Towards unified system modeling with the ModelicaML UML profile. In *1st International Workshop on Equation-Based Object-Oriented Language and Tools*, Berlin, Germany, july 2007.

[3] M. Abadi and B. Blanchet. Analyzing Security Protocols with Secrecy Types and Logic Programs. In *29th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL 2002)*, pages 33–44, Portland, Oregon, January 2002. ACM Press.

[4] M. M. Abdalla, F. Khendek, and G. Butler. New results on deriving sdl specification from mscs. In *Proceedings of SDL Forum'99*, Montreal, Canada, June 1999.

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

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

[7] S.Z. Ahmad. Analyzing Suitability of SysML for System Engineering Applications. In *Master thesis, School of Engineering*, volume 49, pages 627–642. Blekinge Institute of Technology, 2007.

[8] S. Ahumada et al. Specifying Fractal and GCM components with UML. In *XXVI International Conference of the Chilean Computer Science Society (SCCC'07)*, Iquique, Chile, Nov 2007.

[9] Alessandro Aldini and Marco Bernardo. A formal approach to the integrated analysis of security and QoS. In *Reliability Engineering and System Safety*, volume Vol. 92, pages 1503–1520. Elsevier, 2007.

[10] Yomna Ali, Sherif El-Kassas, and Mohy Mahmoud. A rigorous methodology for security architecture modeling and verification. In *Proceedings of the 42nd Hawaii International Conference on System Sciences*, volume 978-0-7695-3450-3/09. IEEE, 2009.

[11] R. Alur, K. Etessami, and M. Yannakakis. Inference of message sequence charts. *IEEE Transactions on Software Engineering*, 29(7):623–633, 2003.

[12] Rajeev Alura, Kousha Etessamib, and Mihalis Yannakakisc. Realizability and verification of msc graphs. *Theoretical Computer Science: Automata, Languages and Programming*, 331(1):97–114, 2005.

[13] D. Amyot and A. Eberlein. An evaluation of scenario notations and construction approaches for telecommunication systems development. *Telecommunications Systems Journal*, 24(1):61–94, 2003.

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

[15] Charles André, Marie-Agnès Peraldi-Frati, and Jean-Paul Rigault. Integrating the synchronous paradigm into UML: Application to control-dominated systems. In Jean-Marc Jézéquel, Heinrich Hussmann, and Stephen Cook, editors, *UML 2002 - The Unified Modeling Language*, volume 2460 of *Lecture Notes in Computer Science*, pages 245–274. Springer Berlin / Heidelberg, 2002.

[16] L. Apvrille and A. Becoulet. Fast and multi-platform prototyping of embedded systems from UML/SysML models. In *The 14th edition of the Sophia Antipolis MicroElectronics Forum (SAME'2011)*, Sophia-Antipolis, France, October 2011.

[17] L. Apvrille and A. Becoulet. Prototyping an embedded automotive system from its UML/SysML models. In *ERTSS'2012*, Toulouse, France, February 2012.

[18] L. Apvrille, S. Coudert, and C. Leduc. A framework for the formal verification of infinite systems. In *The 18th IEEE International Symposium on Software Reliability Engineering (ISSRE 2007)*, Trollhättan, Sweden, Nov 2007.

[19] L. Apvrille, J.-P. Courtiat, C. Lohr, and P. de Saqui-Sannes. TURTLE: A real-time UML profile supported by a formal validation toolkit. 30(7):473–487, Jul 2004.

[20] L. Apvrille and P. de Saqui-Sannes. Adding a methodological assistant to a protocol modeling environment. In *8th annual international conference on New Technologies of Distributed Systems (NOTERE'2008)*, Lyon, France, Jun 2008.

[21] L. Apvrille and P. De Saqui-Sannes. TURTLE: Four weddings and a tutorial. In *Embedded Real Time Software and Systems (ERTS2'2010)*, Toulouse, France, May 2010.

[22] L. Apvrille and P. De Saqui-Sannes. Un assistant méthodologique UML. Modélisation et vérification formelle de protocoles guidées par des patrons. *Technique et Science Informatiques*, 30/3:309–337, March 2011.

[23] L. Apvrille, P de Saqui-Sannes, and F. Khendek. TURTLE-P: un profil UML pour la validation d'architectures. In *Colloque Francophone sur l'Ingénierie des Protocoles (CFIP'2003)*, Paris, France, October 2003.

[24] L. Apvrille, P. de Saqui-Sannes, and F. Khendek. Synthèse d'une conception UML temps-réel à partir de diagramme de séquences. In *Colloque Francophone sur l'Ingéniérie des Protocoles*, Bordeaux, France, March 2005.

[25] L. Apvrille, P. de Saqui-Sannes, C. Lohr, P. Sénac, and J.-P. Courtiat. A new UML profile for real-time system formal design and validation. In *Proceedings of the Fourth International Conference on the Unified Modeling Language (UML'2001)*, Toronto, Canada, October 2001.

[26] L. Apvrille, P. De Saqui-Sannes, and A. Mifdaoui. A UML framework for the dimensioning and formal verification of embedded systems. In *Second annual SAFA workshop*, Sophia-Antipolis, France, September 2009.

[27] L. Apvrille, P. de Saqui-Sannes, R. Pacalet, and A. Apvrille. Un environnement UML pour la conception de systèmes distribués. *Annales des Télécommunications*, 61:11/12:1347–1368, Novembre 2006.

[28] L. Apvrille, P. de Saqui-Sannes, P. Sénac, and M. Diaz. Formal modeling of space-based software in the context of dynamic reconfiguration. In *DAta Systems In Aerospace (DASIA'2001)*, Nice, France, May 2001.

[29] L. Apvrille, P de Saqui-Sannes, P. Sénac, and C. Lohr. Verifying service continuity in a satellite reconfiguration procedure. *Journal of Automated Software Engineering*, 11:2:167–191, January 2004.

[30] L. Apvrille, D. Knorreck, and R. Pacalet. Interactive System Level Debugging of Systems-on-Chip. In *S4D 2010*, Southampton, UK, September 2010.

[31] L. Apvrille, A. Mifdaoui, and P. De Saqui-Sannes. Nouvelle approche TURTLE pour le dimensionnement et la validation de systemes répartis temps réel. In *9th annual international conference on New Technologies of Distributed Systems (NOTERE'2009)*, Montreal, Canada, July 2009.

[32] L. Apvrille, A. Mifdaoui, and P. De Saqui-Sannes. Real-time distributed systems dimensioning and validation: The TURTLE method. *Studia Informatica Universalis*, 8(3):47–69, October 2010.

[33] L. Apvrille, W. Muhammad, R. Ameur-Boulifa, S. Coudert, and R. Pacalet. A UML-based environment for system design space exploration. In *Electronics, Circuits and Systems, 2006. ICECS '06. 13th IEEE International Conference on*, pages 1272 – 1275, Dec. 2006.

[34] Ludovic Apvrille. TTool, an open-source toolkit for the modeling and verification of embedded systems. In *http://ttool.telecom-paristech.fr/*.

[35] Ludovic Apvrille. Contribution à la reconfiguration dynamique de logiciels embarqués temps-réel: application à un environnement de télécommunication par satellite. In *Doctorat de l'Institut National Polytechnique de Toulouse*, June 2002.

[36] Ludovic Apvrille, Pierre de Saqui-Sannes, and Ferhat Khendek. TURTLE-P: a UML profile for the formal validation of critical and distributed systems. *Software and Systems Modeling*, 5:449–466, 2006. 10.1007/s10270-006-0029-5.

[37] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, P.C. Heám O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, and L. Vigneron. The avispa tool for the automated validation of internet security protocols and applications. In *CAV 2005*, volume 3576 of *LNCS*, pages 281–285. Springer Verlag, 2005.

[38] I. Assayad and S. Yovine. A framework for modelling and performance analysis of multiprocessor embedded systems: Models and benefits. In *Proceedings of the 8th conference on Nouvelles Technologies de la Distribution (NOTERE'2007)*, Marrakech, Marocco, June 2007.

[39] K. Avnit and A. Sowmya. A formal approach to design space exploration of protocol converters. In *Design, Automation and Test in Europe Conference and Exhibition, 2009. DATE'09*, pages 129–134, April 2009.

[40] 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.

[41] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.

[42] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In *Lecture Notes on Concurrency and Petri Nets*, pages 87–124. W. Reisig and G. Rozenberg (eds.), LNCS 3098, Springer-Verlag, 2004.

[43] 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.

[44] B. Berthomieu and F. Vernadat. Time Petri Nets Analysis with TINA. In *3rd International Conference on The Quantitative Evaluation of Systems (QEST 2006)*, Edinburgh, Scotland, 2006. IEEE Computer Society.

[45] B. Blanchet. From Secrecy to Authenticity in Security Protocols. In Manuel Hermenegildo and Germán Puebla, editors, *9th International Static Analysis Symposium (SAS'02)*, volume 2477 of *Lecture Notes on Computer Science*, pages 342–359, Madrid, Spain, September 2002. Springer Verlag.

[46] B. Blanchet. Automatic verification of correspondences for security protocols. *Journal of Computer Security*, 17(4):363–434, July 2009.

[47] B. Blanchet and A. Podelski. Verification of cryptographic protocols: Tagging enforces termination. *Theoretical Computer Science*, 333(1-2):67–90, March 2005. Special issue FoSSaCS'03.

[48] Bruno Blanchet and David Pointcheval. The computational and decisional Diffie-Hellman assumptions in CryptoVerif. In *Workshop on Formal and Computational Cryptography (FCC 2010)*, Edimburgh, United Kingdom, July 2010.

[49] 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.

[50] Michele Boreale and Maria Grazia Buscemi. Experimenting with STA, a tool for automatic analysis of security protocols. In *Proceedings of the 2002 ACM symposium on Applied computing*, SAC '02, pages 281–285, New York, NY, USA, 2002. ACM.

[51] J.-M. Bruel, B. Cheng, S. Easterbrook, R. France, and B. Rumpe. Integrating formal and informal specification techniques. Why? How? In *Industrial Strength Formal Specification Techniques, 1998. Proceedings. 2nd IEEE Workshop on*, pages 50 –57, 1998.

[52] Marie-Agnès Peraldi-Frati Charles André, Frédéric Mallet. Multiform time in UML for real-time embedded applications. In *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 232–240. IEEE Computer Society, 2007.

[53] A. Chatelain, Y. Mathys, G. Placido, A. La Rosa, and L. Lavagno. High-level architectural co-simulation using Esterel and C. *Hardware/Software Codesign, 2001. CODES 2001. Proceedings of the Ninth International Symposium on*, pages 189–194, 2001.

[54] H. Chockel and K. Fisler. Temporal modalities for concisely capturing timing diagrams. In *Correct hardware design and verification methods, 13th IFIP WG 10.5 advanced research working conference, CHARME'05*, Saarbrucken, Germany, Oct 2005.

[55] Robert Clark and Ana Moreira. Use of E-LOTOS in adding formality to UML. *Journal of Universal Computer Science*, 6:2000, 2000.

[56] Kevin Compton, Yuri Gurevich, James Huggins, and Wuwei Shen. An automatic verification tool for UML. Technical report, 2000.

[57] J.P. Courtiat, C.A.S. Santos, C. Lohr, and B. Outtaj. Experience with RT-LOTOS, a Temporal Extension of the LOTOS Formal Description Technique. *Computer Communications*, 23:1104–1123, 2000.

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

[59] W. Damm, W. Damm, B. Josko, and A. Votintseva. A formal semantics for a UML kernel language, 2003.

[60] Werner Damm and David Harel. Lscs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19:45–80, 2001. 10.1023/A:1011227529550.

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

[62] Haitao Dan, R.M. Hierons, and S. Counsell. Non-local choice and implied scenarios. In *Software Engineering and Formal Methods (SEFM), 2010 8th IEEE International Conference on*, pages 53 –62, Sept. 2010.

[63] P de Saqui-Sannes, L. Apvrille, C. Lohr, and J.-P. Courtiat. Derniers développements autours du profil UML temps-réel TURTLE. In *Actes de la conférence Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'2004)*, Toulouse, France, March 2004.

[64] P. de Saqui-Sannes, L. Apvrille, C. Lohr, and J.-P.Courtiat. TURTLE : un pont entre UML et RT-LOTOS. In *Actes de la conférence Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'2004)*, Besancon, France, June 2004.

[65] P. de Saqui-Sannes, L. Apvrille, C. Lohr, P. Sénac, and J.-P. Courtiat. UML et RT-LOTOS : Vers une intégration informel/formel au service de la validation de systèmes temps réel. In *Actes du colloque francophone sur la Modélisation des Systèmes Réactifs (MSR'2001)*, Toulouse, France, October 2001.

[66] P. de Saqui-Sannes, L. Apvrille, C. Lohr, P. Sénac, and J.-P. Courtiat. UML and RT-LOTOS: An integration for real-time system validation. *European Journal of Automation (JESA)*, 36:1029–1042, 2002.

[67] P. de Saqui-Sannes, T. Villemur, B. Fontan, S. Mota, M. Bouassida, N. Chridi, I. Chrisment, and L. Vigneron. Formal verification of secure group communication protocols modelled in UML. *Innovations in Systems and Software Engineering*, 6:125–133, 2010. 10.1007/s11334-010-0122-3.

[68] Nico de Wet and Pieter Kritzinger. Using UML models for the performance analysis of network systems. *Computer Networks*, 49:627–642, 2005.

[69] Jérôme Delatour and Mario Paludetto. UML/PNO: A way to merge UML and Petri Net objects for the analysis of real-time systems. In *ECOOP Workshops*, pages 511 –514, 1998.

[70] Michel Diaz. Application of petri nets to communication protocols. *In Petri Nets. fundamental Models, Verification and Applications*, 978-1-84821-079-0:27–39, 1989.

[71] Clare Dixon, Mari-Carmen Fernández Gago, Michael Fisher, and Wiebe van der Hoek. Using temporal logics of knowledge in the formal verification of security protocols. In *Proceedings of the 11th International Symposium on Temporal Representation and Reasoning (TIME'04)*, volume IEEE 1530-1311/04, 2004.

[72] Laurent Doldi. *Validation of Communications Systems with SDL*. Wiley, 2003.

[73] D. Dolev and A.C. Yao. On the security of public key protocols. *IEEE trans. on Information Theory*, 29:198–208, 1983.

[74] B. P. Douglass. *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Addison wesley, 2002.

[75] Michael Drouineaud, Maksym Bortin, Paolo Torrini, and Karsten Sohr. A first step towards formal verification of security policy properties for RBAC. In *Proceedings of the Fourth International Conference on Quality Software (QSIC'04)*, volume 0-7695-2207-6/04. IEEE, 2004.

[76] Sophie Dupuy and Lydie du Bousquet. A multi-formalism approach for the validation of UML models. *Formal Aspects of Computing*, 12:228–230, 2000. 10.1007/s001650070018.

[77] Andy Evans, Steve Cook, Steve Mellor, Jos Warmer, and Alan Wills. Advanced methods and tools for a precise UML panel. In *Proceedings of the 2nd international conference on The unified modeling language: beyond the standard*, UML'99, pages 706–722, Berlin, Heidelberg, 1999. Springer-Verlag.

[78] Stephan Flake and Wolfgang Mueller. A UML profile for real-time constraints with the OCL. In *IN UML'2002 - The Unified Modeling Language. 5th international conference*, pages 179–195. Springer, 2002.

[79] B. Fontan. Méthodologie de conception de systèmes temps réel et distribués en contexte UML/SysML. In *Doctorat de l'Université de Toulouse délivrée par l'Université Paul Sabatier*, January 2008.

[80] B. Fontan, P. de Saqui-Sannes, and L. Apvrille. Synthèse d'observateurs à partir d'exigences temporelles. In *14ème colloque International sur les Langages et Modèles à Objets (LMO 2008), Revue des Nouvelles Technologies de l'Information (RNTI-L-1)*, pages 187–203, Montreal, Canada, Feb 2008.

[81] Martin Fränzle and Karsten Lüth. Visual temporal logic as a rapid prototyping tool. *Computer Languages*, 27:231–47, 2001.

[82] A. Fuchs, S. Gürgens, L. Apvrille, and G. Pedroza. On-Board Architecture and Protocols Verification. Technical Report Deliverable D3.4.3, EVITA Project, 2010.

[83] A. Fuchs, S. Gürgens, R. Rieke, and L. Apvrille. 1st Version Architecture and Protocols Verification and Attack Analysis. Technical Report Deliverable D3.4.1, EVITA Project, 2010.

[84] E. Gamma. *Design Patterns*. Addison wesley, 1995.

[85] 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)*, volume 4590, pages 158–163, Berlin Germany, 2007.

[86] B. Geppert. The SDL Pattern Approach. In *Dissertation, Fachbereich Informatik, Universitat Kaiserslautern*, 2001.

[87] Sébastien Gérard, François Terrier, and Yann Tanguy. Using the model paradigm for real-time systems development: ACCORD/UML. In *in OOIS'02-MDSD. 2002*, pages 260–269. Springer, 2002.

[88] Sébastien Gérard, François Terrier, and Yann Tanguy. Using the model paradigm for real-time systems development: ACCORD/UML. In *in OOIS'02-MDSD. 2002*, pages 260–269. Springer, 2002.

[89] P. Giorgini, F. Massacci, J. Mylopoulos, and N. Zannone. ST-Tool: a CASE tool for security requirements engineering. In *Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on*, pages 451 – 452, Aug.-Sept. 2005.

[90] Stefania Gnesi, Diego Latella, and Mieke Massink. Model checking UML statechart diagrams using JACK. In *The 4th IEEE International Symposium on High-Assurance Systems Engineering*, HASE '99, pages 46–55, Washington, DC, USA, 1999. IEEE Computer Society.

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

[92] A. Le Guennec. Méthodes formelles avec UML : Modélisation, validation et génération de tests. In *Actes du 8eme Colloque Francophone sur l'Ingeénierie des Protocoles (CFIP'2000)*, pages 151–166. Hermes, 2002.

[93] F. Lang H. Garavel, R. Mateescu and W. Serwe. CADP 2006: A toolbox for the construction and analysis of distributed processes. In *Tool presentation at CAV'07 (19th International Conference on Computer Aided Verification)*, Berlin, Germany, 2007.

[94] 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.

[95] 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.

[96] 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.

[97] 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.

[98] 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)*, volume LNCS, pages 121–136. Springer, 2009.

[99] IBM. Tau G2, www-01.ibm.com/software/awdtools/tau. 2009.

[100] M.S. Idrees, Y. Roudier, and L. Apvrille. A framework towards the efficient identification and modelling of security requirements. In *5ème Conf. sur la Sécurité des Architectures Réseaux et Systèmes d'Information (SAR-SSI 2010)*, Menton, France, May 2010.

[101] ISO-LOTOS. A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. In *International Standard 8807, International Organization for Standardization - Information Processing Systems - Open Systems Interconnection*, Geneva, July 1987.

[102] ITU-T. Recommendation Z.100, Specification and Design Language (SDL). 1996.

[103] J. Rilling J. Hassine and R. Dssouli. Timed use case maps. In *System Analysis and Modeling: Language Profiles, 5th International Workshop, SAM 2006*, pages 99–114, Kaiserslautern, Germany, 2006.

[104] Ch. Jaber, A. Kanstein, L. Apvrille, A. Baghdadi, P. Le Moenner, and R. Pacalet. High-level system modeling for rapid HW/SW architecture exploration. In *20th IEEE/IFIP International Symposium on Rapid System Prototyping (RSP'2009)*, Paris, France, June 2009.

[105] Ch. Jaber, A. Kanstein, L. Apvrille, A. Baghdadi, and R. Pacalet. Shared resources high-level modeling in embedded systems using virtual nodes. In *Joint IEEE North-East Workshop on Circuits and Systems and TAISA Conference (NEWCAS-TAISA '09)*, Toulouse, France, July 2009.

[106] Chafic Jaber. High-level soc modeling and performance estimation applied to a multi-core implementation of LTE EnodeB physical layer. In *Ph.D. of Ecole doctorale informatique, télécommunications et électronique of Paris*, September 2011.

[107] C. Jard, J.-F. Monin, and R. Groz. Development of Veda, a prototyping tool for distributed algorithms. *Software Engineering, IEEE Transactions on*, 14(3):339 –352, mar 1988.

[108] Jan Jurjens. UMLsec: Extending UML for secure systems development. In Jean-Marc Jézéquel, Heinrich Hussmann, and Stephen Cook, editors, *UML 2002 - The Unified Modeling Language*, volume 2460 of *Lecture Notes in Computer Science*, pages 1–9. Springer Berlin / Heidelberg, 2002.

[109] Enno Kelling, Michael Friedewald, Timo Leimbach, Marc Menzel, Peter Séger, Hervé Seudié, and Benjamin Weyl. Specification and evaluation of e-security relevant use cases. Technical Report Deliverable D2.1, EVITA Project, 2009.

[110] Torsten Kempf, Malte Doerper, R. Leupers, G. Ascheid, H. Meyr, Tim Kogel, and Bart Vanthournout. A modular simulation framework for spatial and temporal task mapping onto multi-processor soc platforms. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 2*, DATE '05, pages 876–881, Washington, DC, USA, 2005. IEEE Computer Society.

[111] F. Khendek, R. Gabriel, G. Butler, and P. Grogono. Implementability of message sequence charts. In *Proceeedings of the first SDL Forum Society Workshop on SDL and MSC*, Berlin, Germany, July 1998.

[112] D. Knorreck, L. Apvrille, and P. De Saqui-Sannes. TEPE: A SysML language for time-constrained property modeling and formal verification. In *Third IEEE International workshop UML and Formal Methods*, Shanghai, China, November 2010.

[113] D. Knorreck, L. Apvrille, and P. De Saqui-Sannes. TEPE: A SysML language for time-constrained property modeling and formal verification. *ACM SIGSOFT Software Engineering Notes*, 36(1):1–8, January 2011.

[114] D. Knorreck, L. Apvrille, and R. Pacalet. Fast simulation techniques for design space exploration. In *47th International Conference Objects, Models, Components, Patterns*, volume 33, pages 308–327, Zurich, Switzerland, June 2009.

[115] D. Knorreck, L. Apvrille, and R. Pacalet. An interactive system level simulation environment for systems-on-chip. In *Second annual SAFA Workshop*, Sophia-Antipolis, September 2009.

[116] D. Knorreck, L. Apvrille, and R. Pacalet. Demonstration of an interactive system level simulation environment for systems-on-chip. In *10th annual international conference on New Technologies of Distributed Systems (NOTERE'2010)*, Tozeur, Tunisia, June 2010.

[117] D. Knorreck, L. Apvrille, and R. Pacalet. Demonstration of an interactive system level simulation environment for systems-on-chip. In *University Booth Demonstration at Design Automation and Test in Europe (DATE'2010)*, Dresden, Germany, March 2010.

[118] D. Knorreck, L. Apvrille, and R. Pacalet. An interactive system level simulation environment for systems-on-chip. In *Embedded Real Time Software and Systems (ERTS2'2010)*, Toulouse, France, May 2010.

[119] D. Knorreck, L. Apvrille, and R. Pacalet. Demonstration of a coverage driven verification environment for UML models of systems-on-chip. In *University Booth Demonstration at Design Automation and Test in Europe (DATE'2011)*, Grenoble, France, March 2011.

[120] Daniel Knorreck. UML-based design space exploration, fast simulation and static analysis. In *Ph.D. of Ecole doctorale informatique, télécommunications et électronique of Paris*, October 2011.

[121] Daniel Knorreck, Ludovic Apvrille, and Renaud Pacalet. Formal system-level design space exploration. In *New Technologies of Distributed Systems (NOTERE), 2010 10th Annual International Conference on*, pages 1 –8, 31 2010-june 2 2010.

[122] Daniel Knorreck, Ludovic Apvrille, and Renaud Pacalet. Formal system-level design space exploration. *Concurrency and Computation: Practice and Experience*, pages n/a–n/a, 2012.

[123] 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.

[124] S. Konrad and B. H.C.Cheng. Real-time specification patterns. In *Proceedings of the 27th international conference on Software engineering*, pages 372–381, St. Louis, MO, USA, May 2005.

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

[126] P. Kukkala et al. Performance Modeling and Reporting for the UML 2.0 Design of Embedded Systems. In *Proc. of the 2005 International Symposium on System-on-Chip*, pages 50–53, Nov 2005.

[127] 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.

[128] Regine Laleau and Amel Mammar. An overview of a method and its support tool for generating b specifications from UML notations. In *Proceedings of the 15th IEEE international conference on Automated software engineering*, ASE '00, Washington, DC, USA, 2000. IEEE Computer Society.

[129] T. le Sergent, R. Heckmann, and D. Kastner. Methodology and benefist of Timing Verification for Safety-Critical Embedded Software. In *DO-178 White paper*, http://www.esterel-technologies.com/DO-178B/request/whitepaper, 2008.

[130] J.Y. Leboudec and P. Thiran. *Network Calculus*. Springer Verlag LNCS volume 2050, 2001.

[131] T. Lecomte, D. Cansell, and D. Méry. Patrons de conception prouvés. In *Journées NEPTUNE*, May 2007.

[132] 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.

[133] 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.

[134] M.V. Linhares, R.S. de Oliveira, J.-M. Farines, and F. Vernadat. Introducing the modeling and verification process in sysml. In *Emerging Technologies and Factory Automation, 2007. ETFA. IEEE Conference on*, pages 344 –351, sept. 2007.

[135] LIP6. Mutekh. http://www.mutekh.org.

[136] Giuseppe Lipari and Enrico Bini. A methodology for designing hierarchical scheduling systems. *J. Embedded Comput.*, 1(2):257–269, April 2005.

[137] C. Lohr, L. Apvrille, P de Saqui-Sannes, and J.-P. Courtiat. New operators for the TURTLE profile. In *6th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'03)*, Paris, France, November 2003. Springer.

[138] Christophe Lohr. Contribution à la conception de systèmes temps-réel s'appuyant sur la technique de description formelle RT-Lotos. In *Doctorat de l'Institut National Polytechnique de Toulouse*, December 2002.

[139] 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.

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

[141] 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.

[142] Lohrey Markus. Safe realizability of high-level message sequence charts. In *Proceedings of the 13th International Conference on Concurrency Theory*, CONCUR '02, pages 177–192, London, UK, UK, 2002. Springer-Verlag.

[143] Trevor Meyerowitz, Claudio Pinello, and Alberto Sangiovanni-Vincentelli. A tool for describing and evaluating hierarchical real-time bus scheduling policies. In *Proceedings of the 40th annual Design Automation Conference*, DAC '03, pages 312–317, New York, NY, USA, 2003. ACM.

[144] 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.

[145] Antonio Ma na and Gimena Pujol. Towards formal specification of abstract security properties. In *The Third International Conference on Availability, Reliability and Security*, volume 0-7695-3102-4/08. IEEE, 2008.

[146] 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.

[147] Iulian Ober, Susanne Graf, and Ileana Ober. Validating timed UML models by simulation and verification. *Int. J. Softw. Tools Technol. Transf.*, 8(2):128–145, April 2006.

[148] Objectiver. Website. In *http://www.objectiver.com/*.

[149] Peter Ochsenschläger, Jürgen Repp, and Roland Rieke. The SH-Verification Tool. In *Proceedings of the Thirteenth International Florida Artificial Intelligence Research Society Conference*, pages 18–22. AAAI Press, 2000.

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

[151] OMG. UML 2.0 Superstructure Specification. In *http://www.omg.org/docs/ptc/03-08-02.pdf*, Geneva, 2003.

[152] OMG. Omg systems modeling language. In *http://www.sysmlforum.com/docs/specs/OMGSysML-v1.1-08-11-01.pdf*, 2008.

[153] OMG. A UML profile for MARTE, beta 2, www.omg.org. 2008.

[154] 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.

[155] J.C. Roger P. Dhaussy and F. Boniol. Proof units of model formal verification (in French). In *Ingénierie Dirigée par les Modèles (IDM'07)*, Toulouse, France, mar 2007.

[156] G. Pedroza, L. Apvrille, and R. Pacalet. Formal security model for verification of automotive embedded applications. In *The 3nd Annual SAFA Workshop (SAFA'2010)*, Sophia-Antipolis, France, October 2010.

[157] G. Pedroza, M. S. Idrees, L. Apvrille, and Y. Roudier. A formal methodology applied to secure over-the-air automotive applications. In *The 74th IEEE Vehicular Technology Conference: VTC2011-Fall*, San Francisco, USA, September 2011.

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

[159] Andy D. Pimentel, Cagkan Erbas, and Simon Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Trans. Comput.*, 55(2):99–112, February 2006.

[160] Miroslav Popovic. *Communication Protocol Engineering.* Tailors and Francis, 2006.

[161] Nicolas Pouillon, Alexandre Becoulet, Aline Vieira De Mello, Francois Pecheux, and Alain Greiner. A generic instruction set simulator api for timed and untimed simulation and debug of mp2-socs. In *In IEEE Proc. of Rapid System Prototyping 2009*, pages 116–122, Paris, France, 2009. IEEE.

[162] L. Rising. *Design Patterns in Communications Software.* Cambridge University Press, 2001.

[163] 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 Conference and Exhibition, 2008. DATE'08*, pages 780–783, March 2008.

[164] T. Soriano S. Turki and A. Sgaier. Mechatronic systems modeling with sysml: A bond graph addendum for energy analysis. *WSEAS transactions on systems*, 4(5):616–624, 2005.

[165] SAE. The sae architecture analysis and design language (aadl) standard (version 2). In *http://www.aadl.info*, 2009.

[166] T. Schattkowsky et al. A model-based approach for executable specifications on recon figurable hardware. In *Design, Automation and Test in Europe Conference and Exhibition, 2005. DATE'05*, pages 692–697, Nov 2005.

[167] H. Schioler, H. P. Schwefel, and M. B. Hansen. CyNC: a MATLAB/Simulink toolbox for network calculus. In *ValueTools '07: Proceedings of the 2nd international conference on Performance evaluation methodologies and tools*, pages 1–10, ICST, Brussels, Belgium, Belgium, 2007. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[168] Jürgen Schnerr, Oliver Bringmann, Alexander Viehl, and Wolfgang Rosenstiel. High-performance timing simulation of embedded software. In *Proceedings of the 45th annual Design Automation Conference*, DAC '08, pages 290–295, New York, NY, USA, 2008. ACM.

[169] H. Schweppe, M. S. Idrees, Y. Roudier, B. Weyl, R. El Khayari, O. Henniger, D. Scheuermann, G. Pedroza, L. Apvrille, H. Seudié, H. Platzdasch, and M. Sall. Secure on-board protocols specification. Technical Report Deliverable D3.3, EVITA Project, 2010.

[170] Bran Selic. Using UML for modeling complex real-time systems. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, LCTES '98, pages 250–260, London, UK, UK, 1998. Springer-Verlag.

[171] H. Seudié, J. Shokrollahi, B. Weyl, A. Keil, M. Wolf, F. Zweers, T. Gendrullis, M. S. Idrees, Y. Roudier, H. Schweppe, H. Platzdasch, R. El Khayari, O. Henniger, D. Scheuermann, L. Apvrille, and G. Pedroza. Secure on-board architecture specification. Technical Report Deliverable D3.2, EVITA Project, 2010.

[172] Insik Shin, Arvind Easwaran, and Insup Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. *Real-Time Systems, Euromicro Conference on*, 0:181–190, 2008.

[173] SkyEye. Webpage of SkyEye. In *www.skyeye.org/*, 2011.

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

[175] SoCLib. SoCLib: an open platform for virtual prototyping of multi-processors system-on-chip (webpage). In *http://www.soclib.fr*, 2010.

[176] Thalès-Alenia-Space. The maestro european project, mobile applications and services based on satellite and terrestrial interworking. 2006. `http://cordis.europa.eu/fetch?CALLER=PROJ\_ICT\&ACTION=D\&CAT=PROJ\&RCN=71246`.

[177] M. J. Toussaint. A new method for analyzing the security of cryptographic protocols. In *Journal on Selected Areas in Communications*, volume Vol. 11, No. 5. IEEE, June 1993.

[178] Issa Traore. An outline of PVS semantics for UML statecharts. *Journal of Universal Computer Science*, 6:2000, 2000.

[179] Denis Trcek and Borka Jerman Blazic. Formal language for security services base modelling and analysis. In *Elsevier Science Journal, Computer Communications*, volume Vol. 18, No. 12. Elsevier Science, 1995.

[180] N. Kicillof V. Braberman and A. Alfonso. A scenario-matching approach to the description and model-checking of real-time properties. *IEEE Transactions on Software Engineering*, 31(12):1028–1041, December 2015.

[181] A. van Lamsweerde. *Goal-Oriented Requirements Engineering: System Objectives to UML Models to Software Specifications*. Wiley, 2006.

[182] Peter van Stralen and Andy D. Pimentel. Trace-based scenario database for high-level simulation of multimedia mp-socs. 2010.

[183] Y. Vanderperren and W. Dehaene. UML 2 and SysML: an approach to deal with complexity in SoC/NoC design. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 716 – 717 Vol. 2, march 2005.

[184] 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.

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

[186] 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 and Test in Europe Conference and Exhibition, 2009. DATE'09*, pages 226–231, April 2009.

[187] 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 Conference and Exhibition, 2006. DATE'06*, 1:1–6, March 2006.

[188] 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.

[189] Kelly Waters. *All About Agile: Agile Management Made Easy!* CreateSpace Independent Publishing Platform, 2012.

[190] P. Wodey, G. Camarroque, F. Baray, R. Hersemeule, and J.-P. Cousin. LOTOS code generation for model checking of STBus based SoC: the STBus interconnection. *This paper appears in: Formal Methods and Models for Co-Design, 2003. MEMOCODE '03. Proceedings. First ACM and IEEE International Conference on*, pages 204–213, June 2003.

[191] 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.

[192] T. Zheng. Validation and refinement of timed MSCs. In *PhD Thesis, Concordia University, Montréal, Canada*, January 2004.