Unité Mixte de Recherche UJF - CNRS - Grenoble INP
Centre Équation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 41
fax : +33 456 52 03 44
http://www-verimag.imag.fr

UNIVERSITÉ DE GRENOBLE

Habilitation à Diriger des Recherches

Spécialité Informatique

Présentée et soutenue publiquement par

Matthieu Moy

13 mars 2014

Modélisation à haut niveau d'abstraction pour les systèmes embarqués

High-level Models for Embedded Systems

JURY

Frédéric Pétrot	Professeur à Grenoble INP, France	Président
Gérard Berry	Professeur au Collège de France	Rapporteur
Rolf Drechsler	Professeur à l'Université de Brême, Allemagne	Rapporteur
Marco Roveri	Senior Researcher, Fondazione Bruno Kessler, Italie	Rapporteur
Samarjit Chakraborty	Professeur à l'université technique de Munich, Allemagne	Examinateur
Benoît Dupont de Dinechin	Directeur Technique, Kalray, France	Examinateur

BibTEX:

```
@PhdThesis{hdr-moy,
  author =  {Matthieu Moy},
  title =  {High-level Models for Embedded Systems},
  school =  {Universit\'e de Grenoble},
  year =  {2014},
  url =  {http://www-verimag.imag.fr/~moy/?Habilitation-a-Diriger-des},
  type =  {Habilitation \`a Diriger des Recherches ({HDR})},
  address =  {Verimag},
}
```

Foreword

This document is my "habilitation" manuscript ("Habilitation à Diriger des Recherches", or HdR in French). It is a summary of the research I did on high-level modeling of embedded systems in the past decade.

The document tries to find the right trade-off between a short summary and an exhaustive and detailed document following two principles. First, verbatim reuse of the content of existing is avoided. Instead of repeating what has already been published, I try to give a different point of view, favoring examples over the general case, intuition over formalization, and figures over textual explanations. Hopefully, this should motivate the reader to search for more details in the existing publications. On the other hand, each contribution is presented with enough detail to let the reader understand the principles, intuition and design choices behind the approaches.

Each of my publications and supervised students/post-docs is cited in the document. The reader can find more information about them in the references at the end of the document (chapter 7). More details about myself and the collaborative projects I've been involved in are available in a separate file.

The document is best viewed in colors, A4 paper, but a black and white print is also readable.

Enjoy your reading!

Acknowledgments

This document summarizes almost 12 years of my career. The list of people who should be thanked for their positive impact on these 12 years is long, and I will undoubtedly forget many of them.

There are short-term "thank you" for people who contributed to my habilitation document and defense. Thanks to Gérard Berry, Rolf Drechsler and Marco Roveri for the time they spent reviewing the manuscript; to Samarjit Chakraborty and Benoît Dupont de Dinechin for their role as examiners and to Frédéric Pétrot for accepting to be president of my Jury.

I would also like to thank people who made the day of the defense such a pleasant experience. Those who made videoconferencing possible, those who took care of the buffet, those who attended the talk, and those who enjoyed the Crémant and the macarons with me afterwards.

Thanks to people who helped me to prepare the habilitation, for their comments on drafts of the manuscript and during defense rehearsals.

There are also many long-term "thank you", for all the colleagues with whom I interacted the last 12 years, and who make me what I am today. To Oded Maler, who gave me a first taste of what research is, and then to Florence Maraninchi and Laurent Maillet-Contoz who supervised my Ph.D, and initiated the STMicroelectronics/Verimag collaboration which guided most of my works. To all the colleagues with whom I co-supervised students or co-authored articles. And of course, a warm "thank you" to all students and post-docs I supervised.

Finally, thanks to my girlfriend.

Contents

Chapter 1

Introduction

> *There are no airplanes, only computers that fly. There are no cars, only computers we sit in. There are no hearing aids, only computers we put in our ears.*
>
> — Cory Doctorow, *The coming war on general computation (2011)*

This document summarizes my research over the last decade. The research was done in the Verimag laboratory. I will recall the work done during my Ph.D (2002-2005), and present the one done as a permanent member, since 2006. I contributed to several areas related to high-level modeling of embedded systems. The contributions presented in this document can be classified according to several criteria: functional properties (does the system produce the right result?) or non-functional ones (does it produce the result at the right time? How much resources does it use?); formal models or simulation-based approaches; targeting software or hardware... The contributions are summarized below in Section 1.2.

A large part of the research deals with virtual prototypes for Systems-on-a-Chip and was done in close collaboration with STMicroelectronics. The collaboration was initiated by my Ph.D, and continued since then with 2 collaborative projects, 3 joint Ph.D, plus the OpenES project which just started. This industrial partnership naturally implies a pragmatic approach: we have to take into account existing tools and workflows, not just propose new ones. In this case, the standard is SystemC/TLM, a set of libraries on top of C++. As a consequence, we have to deal with the complexity of C++.

1.1 Embedded systems

Embedded systems can be defined as computing systems that are not computers. Interestingly, while it seems natural to associate computer science with computers, only 2% of processors today are used in general-purpose workstations [Har04].

The evolution of general purpose computers and the one of embedded systems have both been following Moore's law, leading to an amazing complexity of hardware in terms of number of transistors and hardware components. One point that distinguishes most embedded systems is that their complexity is usually exposed to the programmer: while each generation of Pentium processor is more complex than the previous one, porting software to new hardware is usually as simple as a recompilation, if at all needed. On the other hand, most embedded systems come with a set of sensors and actuators, and have an architecture that is at least partially visible to the software.

Because of the diversity of embedded systems, it is hard to make any general statement about "embedded systems" without refining the definition. In this document, "embedded systems" will usually refer to consumer electronic devices, with an emphasis on complex systems.

1.1.1 Complexity of Hardware

The need for performance optimization (both to provide more computing power and to consume less energy) has led to architectures where the functionality is split across multiple components. In a typical system, the non-performance critical parts can be implemented in software, but the intensive computations are delegated to some specific hardware. One option is to use a fully specialized hardware component (*IP* blocks, for *Intellectual Property*), that the software will have to program properly. A dedicated IP is more efficient both in terms of power consumption and speed than a software implementation.

Another trend is to use *many-core* accelerators like STMicroelectronics' *STHorm* (formerly known as *Platform 2012*) [MBF+12] or Kalray's *Multi-Purpose Processor Array (MPPA)* [Kal12]. These many-core systems break the common "shared memory SMP" programming model. First, the number of processors is too high to allow efficient cache-consistency, so the processors in the array use local memories and explicit transfers, usually using dedicated DMA components, to the global memory. Then, the high processor density is only possible because the individual processors are simple enough. For example, Kalray's MPPA 256 counts 256 cores (with plans to increase to 1024), plus 16 cores for the I/O subsystems, using its own proprietary core technology. STHorm is made of clusters of 16 STxP70 processors. Such processors typically cannot run a general purpose operating system; e.g., Kalray's MPPA cluster cores work non-preemptively. This is acceptable for very specialized computation, but unacceptable for high-level application code, hence the full system should contain at least one more general purpose processor. The result is a highly heterogeneous system, with several pieces of software compiled for different processors and possibly different operating systems.

1.1.2 The Importance of Software

While dedicated hardware is essential for performance, there is also a number of good reasons to move the complexity of the system to the software part. The programming models available in software developments are much richer and allow solving algorithmically hard problems in fewer lines of code (see for example how easy it is to do dynamic memory allocation or recursive calls in software). Many embedded systems run an operating system like Linux, which can manage concurrency between applications, resource sharing and easy access to the underlying hardware.

A key advantage of software is that it brings a lot of flexibility since it is possible to update it at any stage of the system's life-cycle (unlike fixing hardware bugs, which may require re-fabrication of the masks and cost millions of dollars before shipping the first fixed chip!). Unless a hardware implementation can be completely trusted, a software implementation is usually preferred for any critical part of the system.

As a result, systems are composed of a large number of hardware and software components that depend on each other. It is not sufficient to validate each component individually to ensure the correct behavior of the system. Both the architectural choices and the functional behavior need to be validated at the *system-level*, i.e., considering the whole hardware platform together with the software that runs on it.

1.1.3 Constrained Systems

Interaction with the environment enforces constraints on the design of a system. Most embedded systems are real-time systems: either *soft real-time* systems, for which the interaction with the environment is supposed to be performed at the right time or *hard real-time*, for which the interaction needs to be performed at the right time. In consumer electronics, systems are usually soft real-time (for example, a multimedia application will produce unpleasant glitches, or sound/image desynchronization if it is badly timed).

Other *non-functional properties* like silicon area, power consumption and temperature can also become very important. For mobile devices, the power consumption obviously impacts the battery life, but a high power consumption is not always acceptable even for permanently plugged devices. Not only power consumption has an environmental impact, but it also has a financial cost: a chip that consumes too much requires a more expensive packaging and may need a cooling system, possibly noisy for the user. Worse, power consumption peaks can lead to overheating that can physically damage the chip, or to undervoltage that can break the functional behavior. In general, bad thermal behavior (high temperature or temperature gradient) lead to faster aging of the silicon.

Power-saving can be done at various levels. Obviously, optimizing the silicon electrical conductivity and transistor capacitance at the lowest, physical level can change the consumption of the chip (e.g., using FD-SOI [His01] instead of bulk CMOS). Optimizations can also be done within synthesizers, to minimize bit flips for a given computation, or to reduce silicon area. But a large part of optimization opportunities in modern systems is found at the system-level: a global *power management* policy should be able to switch off components that are not used and to reduce the voltage to the minimum required to accomplish the currently running tasks. Reducing the voltage saves energy but implies reducing the frequency hence slowing down the computation.

In traditional computers, these constraints can often be managed with a *best-effort* policy, i.e., make computers "as fast as possible" and consume "as little power as possible". This is generally not possible with embedded systems: over-dimensioning the design to meet the constraints may not be acceptable given the competitivity of the domain, and an under-dimensioned system may be unusable (e.g., a set-top box that is not able to deliver the video stream in real-time is basically useless). It is therefore important not only to be able to optimize these non-functional properties, but also to be able to evaluate them and validate the system taking them into account as early as possible in the design flow.

1.1.4 Modeling and Virtual Prototyping

A common practice is to write *models*, i.e., simplified versions of the real system, before the system is available. There are many kinds of models, depending on the intended application. In *model-driven development*, models are used as the starting point of the implementation flow, using automatic or semi-automatic code generation tools. But not all of them are used to derive an implementation. Models of the physical environment can be used for testing or verification, but obviously won't be embedded in the real system. A thriving trend is *virtual prototyping*, where an executable model of the system is developed early in the design flow to validate some architectural choices or for embedded software development, but not directly to derive an implementation.

Virtual prototyping allows activities that were previously performed on the real system to be performed in a virtual, i.e., software-simulated, platform. Virtual prototypes can be seen as executable specifications of the system. They may be used as reference models for validation of the real system, but are usually not meant to be the starting point of automatic synthesis. Instead, a manual refinement or rewrite has to be performed.

The evolution of programming languages has allowed programmers to abstract away many hardware specificities. From binary machine code to assembler, and then to imperative languages like C, compilers have allowed abstracting away the instruction coding. Some modern languages allow even more distance between the code written and the way it will run on the machine.

For example, the *Lustre* [BCH+85] programming language allows the programmer to write a highly parallel, synchronous, and data-flow description of the system, and lets the compiler do the job of statically scheduling the operations and generating the imperative code. At this level,

one can either refer to the description as "programs", since it is executable and can be compiled to assembly code, or as "models", since it allows abstract reasoning, and even formal proofs (e.g., Lesar [HLR92] or Nbac [Jea03]). In the work presented below, Lustre is used either as an intermediate formal language (Sections 3.2.1 and 3.2.3) or as a modeling language (Section 5.3). We also used Lustre as an implementation language for a toy NXT-robot controller with Valentin Bousson.

The domain of hardware design has followed a similar evolution: from gate-level to RTL, synthesizers have allowed abstracting away the implementation details of state-machines and registers, and modern *high-level synthesizers* are able to generate hardware from simple enough sequential C code [GDC92].

These approaches are appealing, but not always applicable. The subset of C accepted by high-level synthesizers is much smaller than the one developers would like to use for very high-level modeling. Dedicated languages are purposely limited to prevent dangerous constructs like dynamic memory allocation. Most of the work presented in this report focuses on models that are not meant to be used directly for implementation. An exception is the work done on abstract interpretation, which applies directly to software implementations, not to models or prototypes. The following section presents several categories of approaches, and classifies the contributions of the document in these categories.

1.2 Contributions to Various Model-Based Approaches

1.2.1 Virtual Prototyping and Simulation of System-on-Chips

The *Register Transfer Level (RTL)* abstraction level is the entry point of the hardware synthesis flow. While more abstract than gate-level designs, RTL programs still require the developer to describe all the micro-architecture of the platform, i.e., describe each register, each pipeline stage, each communication element and each state-machine behavior at each clock tick. As a result, RTL programs simulate very slowly, and the simulation speed is inversely proportional to the size of the platform. For large systems, RTL simulation is applicable on individual components, but far too slow for a day-to-day use at the system-level (for example, booting an operating system can take tens of hours at this level [HYH$^+$11]).

Various techniques have been proposed to speed up RTL simulations. Optimizers like Verilator [Sny12] or Carbon Model Studio [CDS13] can generate higher-level programs from synthesizable code, achieving an important simulation speedup. Performance-consuming parts of the RTL system can be replaced by a faster simulator: typically an RTL processor can be replaced by a faster *Instruction Set Simulator (ISS)*, which interprets the software's executable binary without simulating all the details of the processor. These techniques provide a valuable speedup, but are far from sufficient to turn a 10 hours simulation into a delay short enough to be acceptable in the typical "compile/debug&execute" cycle of a software developer. We need several orders of magnitude of speedup, and cannot hope for such a speedup without a major change in the level of abstraction.

The *Transaction-Level Modeling (TLM)* [Ghe05, Ope08] level of abstraction was introduced in response to this problem. One can find many definitions of TLM, and even the definition standardized by IEEE and the Accellera Systems Initiative (ASI, formerly OSCI) consortium is declined in several flavors. A first, simple definition can be given as "a model in which we describe everything that is needed for the software to execute, and only this".

In this context, "software" includes low-level software such as drivers, so the definition implies that the address map of the platform be modeled properly. This in turn implies that the overall architecture of the platform be reflected in a TLM model, including register sets. Also, a notion of concurrency must be visible in the model, since the software can run scenarios like "program a

computation in component A, then program component B, and then get the result of component A", which requires *concurrency* between components A and B.

On the other hand, many details that appear in an RTL design are abstracted away in TLM. For example, the bus protocol is abstracted into a very simple interface based on function calls; in short, the initiator components call functions of the target components similarly to what is done in remote-procedure call frameworks. The micro-architecture within components can also be abstracted away. For example, from the software point of view, it is sufficient to know that a processor executes an instruction set, but the pipeline and cache need not be visible for the software to execute.

The industry-standard for TLM is *SystemC* [Ope11], which adds to C++ the missing elements for TLM. The first addition is a notion of simulated concurrency. Although SystemC programs are usually executed sequentially, the system they represent can be parallel. A SystemC program contains a set of processes that are executed in interleaving semantics by a scheduler. SystemC provides a notion of *simulated time*: concurrency between two processes is simply modeled by the fact that they execute at the same simulated time.

SystemC was originally designed for levels of abstractions lower than TLM. In particular, the communication mechanisms were directly inspired from the VHDL and Verilog ones (e.g., the `sc_signal` with δ-cycle mechanism). SystemC 2.0 introduced the notion of user-defined communication channels, allowing users to define their own higher-level channels. Based on this infrastructure, different companies started developing in-house protocols for TLM. During my Ph.D, I could follow the birth and evolutions of the *TAC* protocol [ST05] proposed by STMicroelectronics, while for example, GreenSoCs which was both a partner and a competitor was developing GreenBus [Gre07]. Then started the standardization process, required for interoperability. Most participants were pushing the solution they had already deployed internally. A first compromise was TLM-1.0, in 2005, which standardized the principle of communication based on function call, but left the argument and return type of this function (`tlm_transport_if::transport()`) unspecified. The way data was exchanged was described, but the content of transaction was not. Therefore, the standard provided no complete solution to the interoperability problem. The standardization process continued with TLM-2.0 [Ope08], 3 years later, where the content of transaction (*payload*) was standardized. Finally, the TLM-2 OSCI standard was incorporated in the SystemC IEEE standard in its latest version (1666-2011).

Essentially, a transaction transports an address, a piece of data of arbitrary length, a command (e.g., read, write, ...) and a status (ok, error, ...). Additional fields are provided to allow different modeling styles and optimizations.

Figure 1.1 shows a graphical view of an example TLM platform. The example is extracted from a lab-work given to Ensimag students in my SystemC/TLM course [Moy12a]. The actual platform is a small system running on an FPGA and executing Conway's Game of Life. It contains a processor, a timer (to control when a new image should be computed), a VGA video controller, a *General Purpose Input-Output (GPIO)* component to manage buttons for user-interaction, a *Universal Asynchronous Receiver Transmitter* (*UART*) that can send data to a computer over a serial link, an *InTerrupt Controller (ITC)*, and two RAMs. The TLM platform is very simple (it fits in 1000 lines of C++ code), especially when compared to the RTL one (around 10,000 lines of VHDL). However, following the principle presented above, the platform contains just enough detail to execute software, and the same C code can run on the real system and on the TLM platform.

Each box on Figure 1.1 corresponds to a SystemC component, which contains functions to be called through their target socket, some active process code, or both. Most communication is done through a shared bus, modeled by a TLM-2 channel. Interrupts (IRQ) do not need the expressiveness of TLM-2, as they transport no data nor address. They are modeled as simple SystemC signals.

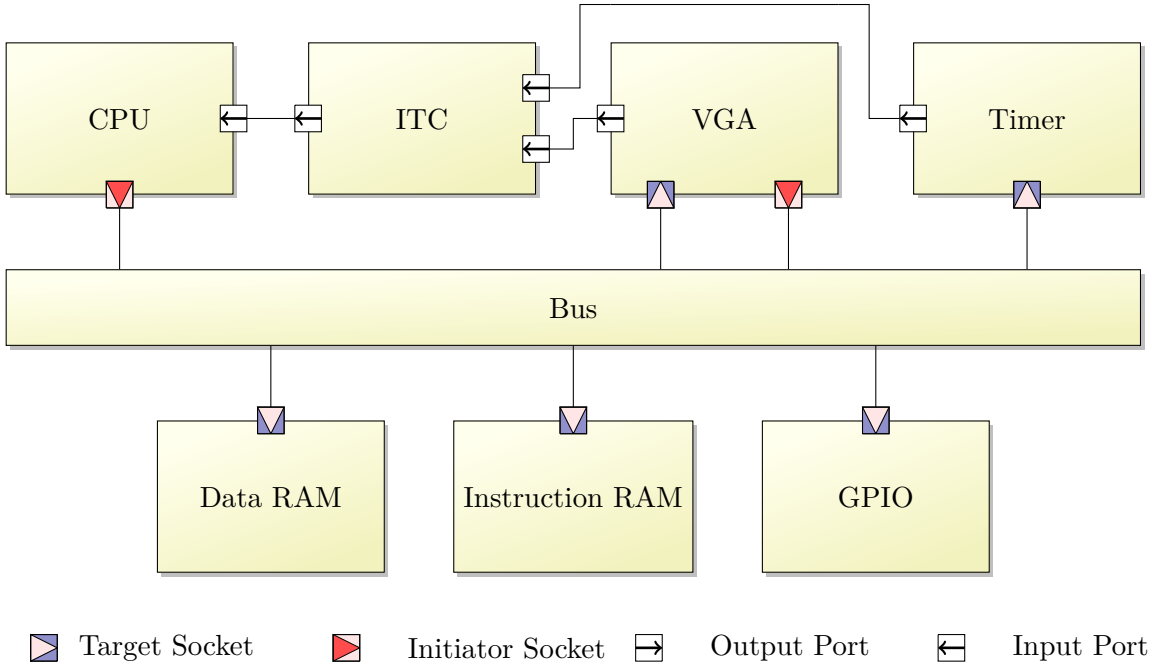


Figure 1.1: Graphical View of an Example TLM Platform

Contributions in the Area of Virtual Prototyping

We made the following contributions to the area of virtual prototyping with TLM:

- We developed dedicated compilation techniques for SystemC. SystemC cannot be considered as a "normal" programming language [MMK10], since it is implemented as a library for C++, and the equivalent of a compiler front-end should be able to extract information that are built during the execution of the simulator, before the actual simulation starts. We present techniques implemented in the tool *PinaVM* [MM10b] that extract both the behavior of processes and the architecture of the platform, and is able to link these together. This contribution will be presented in Section 3 and has been published in [MMK10, MM10a]. It is a continuation of the work done during my Ph.D [MMMC05b, MMMC06, Moy05b] which will be briefly recalled too.

- Based on this tool, we developed an *optimizing compiler*, dedicated to SystemC. This compiler uses the information provided by the front-end PinaVM to perform optimizations like function inlining across an interconnect, hence requires a visibility on the architecture and a knowledge of SystemC's semantics. This contribution will be presented in Section 3.3.

- We identified some limitations of the traditional model of time and concurrency in SystemC/TLM with respect to faithfulness. In particular, we show that some memory-model related bugs cannot be discovered with existing techniques. This contribution is presented in Section 4.1.1 and has been published in [FMMCM11].

- We developed a new TLM simulator called *jTLM* (for "Java TLM"). This simulator was initially created to study the TLM level of abstraction without the constraints of SystemC, and was then used as an experimentation framework to develop new features like the

notion of *task with duration*. Tasks with duration allow a more efficient parallelization and partially solves the faithfulness issue presented above. This contribution will be presented in Section 4.1.4 and has been published in [FM11b, FM11a].

- The notion of tasks with duration developed in jTLM was then adapted to SystemC, providing the same benefits, but without the need for a specialized scheduler. The result is the `sc-during` library [Moy12b, Moy13]. It will be presented in section 4.1.5.

- We contributed to the development and validation of a "Smart FIFO" [HCG+13], a model of a FIFO component which minimizes context-switches using temporal decoupling. Unlike most approaches, the Smart FIFO guarantees that the behavior obtained with temporal decoupling is equivalent to a behavior produced by a non-decoupled model.

- We developed new approaches and tools for estimation of non-functional properties like *power consumption* and *temperature* at different levels of abstraction. These tools allow co-simulating a functional model with a non-functional solver that models power consumption and temperature dissipation. The cosimulation interface [BMM+13b] is very general, but its application to loosely timed systems requires some work to avoid unrealistic results like spurious temperature peaks in simulation when they would not happen in the real system [BMM13a]. This contribution will be presented in section 4.2.4.

1.2.2 Formal Verification of Functional Properties

Virtual-prototyping and simulation-based approaches have had a number of success stories in the industry. However, an obvious limitation of these approaches is that the validation is done on a finite number of executions. It can be desirable to get more formal guarantees about the system, hence to work with more formal models. Formal verification is essential for critical embedded systems, and may also increase the reliability or reduce the testing effort of non-critical systems.

While simulation-based approaches are inherently limited in scope, formal approaches are limited in theory by Rice's theorem, stating that no non-trivial analysis method can be at the same time:

1. Automatic: in the form of an algorithm that terminates in finite time without human intervention,
2. Sound: results provided by the analysis are proved to hold on the system,
3. Complete: any property that holds can be exhibited by the analysis,
4. Unbounded: not limited to bounded execution or state-space.

Relaxing any of these conditions allows an analysis to be performed, and indeed correspond to one domain of software verification. Relaxing constraint 1 (*automation*) is done by proof-assistants that require the intuition of a human user to carry out the proof. Bug-finding techniques (e.g., testing, or lint-tools that only apply heuristics to find common bad patterns in the code) may not be *sound* (constraint 2) in the sense that a system validated with testing or linting has no guarantee to be actually correct. Relaxing constraint 3 (*completeness*) leads to techniques that may actually prove properties, but may be unable to answer (i.e., answer "yes" or "don't know" when asked to prove a property). One approach in this category is abstract interpretation, developed below. Relaxing the last constraint (4, *boundedness*), the state-space becomes finite and can be exhaustively explored, which is done in *model-checking* approaches. The boundedness can be applied on the length of execution traces, or on the size of the state-space. In the first case, the term *Bounded Model-Checking (BMC)* is used. BMC can formally prove that no property violation can occur in less than N steps, but cannot prove properties valid for any execution. A common technique in bounded model-checking is to encode an execution trace of size N, starting from an initial state and ending in an error state, into a logic

formula, and then check for the satisfiability of this formula. If the formula is satisfiable, then a model of this formula represents a counter-example. If not, the property is proved for any execution of size N. If the formula is purely Boolean, the satisfiability problem is called *SAT*. If the formula is not purely Boolean, then SAT-solving techniques can be combined with an external theory to form a *Satisfiability Modulo Theories (SMT)* solver.

Abstract interpretation is a theory that allows the automatic, sound analysis of a possibly unbounded system (e.g., a system that contains numerical variables). Abstract interpretation associates a set of possible variables valuations with each program point. The concrete set of variable valuations being potentially infinite, it has to be abstracted in a so-called *abstract value*, which has a finite machine representation. The set of possible abstract values is called the *abstract domain*. Different abstract domains provide different precision and algorithmic costs. For example, the domain of intervals allows an algorithmically cheap analysis, but cannot represent relations between variables. The polyhedra domain is more expressive and usually provides more precise results, but most operations are exponential in the number of variables.

The analysis propagates abstract values from program points to program points until a fixpoint is reached. The termination of the iteration can be ensured by a widening operator. Figure 1.2 illustrates this on a very simple example.

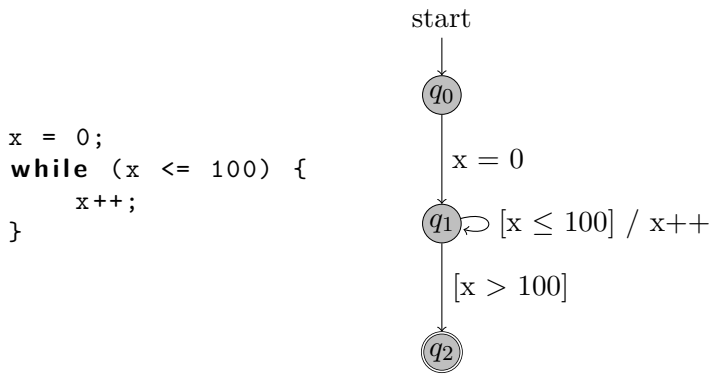


Figure 1.2: Traditional Abstract Interpretation on a Simple Example

The analysis associates a set of possible values $V(q)$ for the variable `x` with each program point q (in this case, control-points of the control-flow graph). Initially, we start with an under-approximation of the possible valuations: $V(q_0) =]-\infty, +\infty[$ and $V(q_1) = V(q_2) = \emptyset$. Then, the analysis propagates the values on the transition $q_0 \rightarrow q_1$, yielding $V(q_1) = \{0\}$. The self-loop $q_1 \rightarrow q_1$ can then be applied, providing successive values $[0,1], [0,2], \ldots$ for $V(q_1)$. The succession of values could be large, or even infinite, so the widening operator is used to enforce and speed up convergence. For example, after computing $[0,1]$ and $[0,2]$, the widening operator can interpolate the set directly to $V(q_1) = [0, +\infty[$. At this point, the transition $q_1 \rightarrow q_2$ become fireable, yielding $V(q_2) = [0, +\infty[\cap]100, +\infty[=]100, +\infty[$. After this, firing any transition does not exhibit any new value: a fixpoint of the iterations with widening is reached and the valuations we obtained provide a safe invariant for the program. We can further tighten the invariant with *narrowing* iterations: perform one or several more iterations without widening, and without performing a union with the previous invariant candidate. The narrowing iterations can recover part of the precision that was lost by widening.

Contributions in the Area of Verification

We made the following contributions related to formal verification:

- We developed several verification tools for sequential programs. Some of the tools are merely reproduction of traditional verification techniques (abstract interpretation, model-checking). The tool PAGAI, on the other hand, first implemented state-of-the-art static analysis using abstract interpretation together with SMT solving, and then allowed proposing new techniques that combined and improved the existing ones. PAGAI also allowed a comparison of existing and new techniques on real-life programs. These analyzers will be presented in Section 2 and have been published in [HMM12b, HMM12a].

- We developed verification tools dedicated to SystemC. Borrowing some ideas from the LusSy [MMMC05a, Moy05a, Moy05b, MMMC06] tool, developed as part of my Ph.D, we provided new techniques to exploit co-routine semantics of SystemC, and the properties of the Static Single Assignment (SSA) form used as intermediate format in modern compilers for formal verification. This contribution will be presented in Section 3.2 ; it led to the publications [BGM+09b, TCMM07, MMJ11, MMC+08].

1.2.3 Formal Models for Non-Functional Properties

Simulation-based approaches can also be insufficient to enforce non-functional properties. Although functional correctness is often more important than non-functional properties (e.g., a slow computer is preferred to a computer that crashes periodically), embedded systems can also require that some non-functional properties be formally verified.

For example, in a hard real-time system, strong guarantees are needed on the *Worst-Case Execution Time (WCET)* of a task, or on the *Worst-Case Traversal Time (WCTT)* of an event to process in a data-flow system. Doing formal WCET analysis on a TLM model for a complex hardware/software system would not be possible, as it would require analyzing the product of a relatively complex and detailed hardware model, by the compiled software.

Formal models are needed to conduct a formal proof, but such models cannot be automatically extracted from the implementation. This differs from formal verification presented above where the formal model on which the analysis runs is usually extracted automatically from the program under analysis.

Part of this document describes the work carried out in formal models for performance analysis. The starting point is the *Modular Performance Analysis (MPA)* with *Real-Time Calculus (RTC)* [TCN00] developed by Lothar Thiele *et al.* based on the Network Calculus [LBT01] theory. The idea is to model a system with a set of components communicating together through event streams, as sketched in Figure 1.3. The actual data exchanged between components is abstracted away; since events contain no information, one can only count the number of occurrences of an event over a time interval. Figure 1.3 shows only the events to be processed, but the MPA framework is more general and allows modeling computing resource in the same way. The analysis is then done by manipulating abstractions of event streams called *arrival curves*, which specify bounds on the number of events that can arrive during any time interval. In Real-Time Calculus, arrival curves are usually manipulated by pairs: the lower curve (α^l) specifies a lower bound and the upper curve (α^u) an upper bound. For example, a pair of arrival curves can express: "between 1 and 5 events can arrive per second, and no more than 10 events can arrive during a period of 5 seconds".

The analysis in MPA is done component per component: the arrival curves of output streams for each component are computed as functions of the input arrival curves. In Figure 1.3, for example, the analysis for Component_1 applies the function F_1 to the input arrival curve α_1 to compute an output arrival curve α'_1 (by convention, input curves are usually denoted as α and output as α'). This output arrival curve is in turn used as the input for the second component (i.e., $\alpha_2 = \alpha'_1$). If the system is composed of N components, the analysis will require

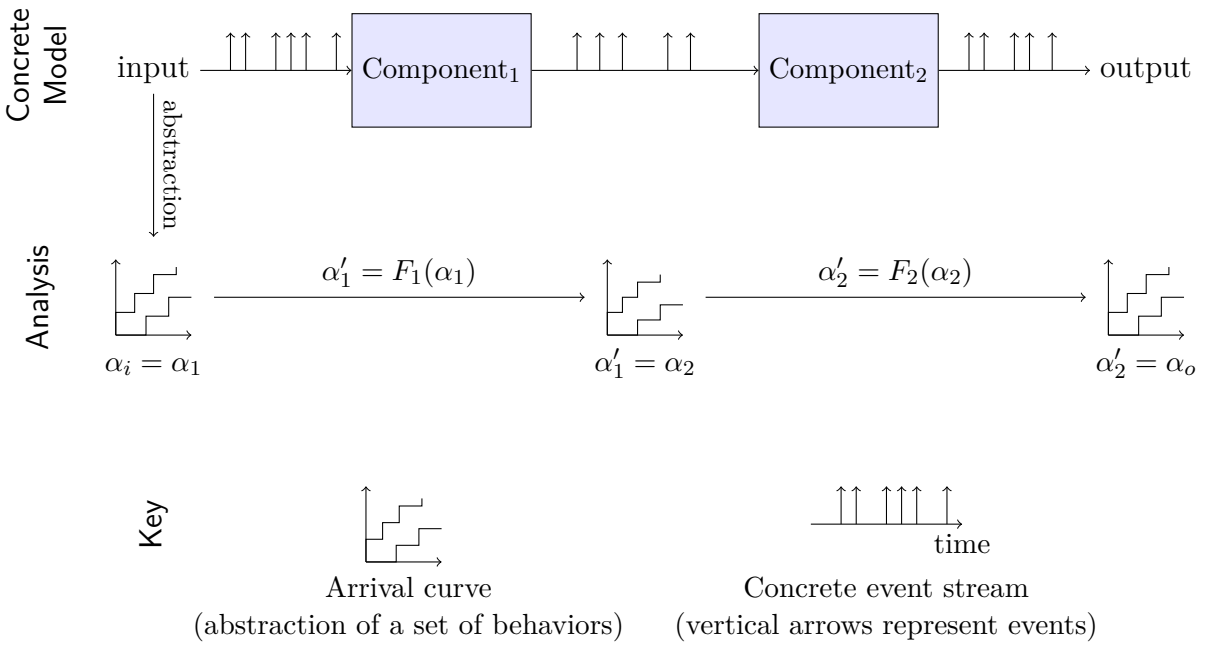


Figure 1.3: Modular Performance Analysis

N computations on arrival curves. In case of cyclic dependencies, a fixpoint can be computed iteratively [JPTY08].

Originally, the analysis of a component within MPA was done in a purely analytic manner, using Real-Time Calculus formulas. When the component is simple enough (e.g., greedy processing element, which processes events as fast as possible given the available resource, and stores unprocessed elements in a FIFO buffer), this approach can provide optimal hard bounds, for a low algorithmic complexity. Unfortunately, a limitation of pure RTC is that it cannot handle the notion of state within a component. Our goal when we started using MPA was to handle performance in a more general way than only timing, in particular to be able to deal with power consumption. Before thinking about extending RTC with power consumption properties, the very first step was to deal with the timing of power-managed components, i.e., components that can be switched off or put in some degraded mode to reduce their consumption. This kind of state-based specification cannot be handled directly by Real-Time Calculus.

An alternative to pure RTC is to analyze individual components using more expressive analysis technique, typically using an automata-based formalism and some state-space exploration techniques such as model-checking. This can be done within the MPA framework if the state-based analysis can use arrival curves as input, and produce arrival curves. This kind of approach has been experimented with a dedicated automaton formalism called *event-count automata* [PCTT07], and adapted to traditional *timed automata* [Upp07, LPT09, LPT10]. The idea is to write adapters from RTC to the automata-based formalism: on the way out of RTC, the arrival curves are compiled into automata that produce the set of event streams specified by arrival curves, and on the way back to RTC, an observer computes the minimum and maximum number of events over a time interval. The analysis of one component can be done using expensive model-checking algorithms, but it should be noted that the analysis of individual components is done individually, hence the overall complexity still grows linearly with the number of components, although it may grow exponentially with their size.

Contributions in the Area of Non-Functional Properties

The contributions to the Modular Performance Analysis theoretical framework improve the connection from RTC to state-based formalisms following two directions:

- We provide two new connections from RTC to state-based formalisms and tools. We improve existing approaches for the connection to timed automata and the Uppaal [LPY97] model-checking tool, allowing some granularity-based abstractions for better performance, and provide a connection to the Lustre language, which allows using abstract interpretation and Satisfiability Modulo Theory (SMT) tools to analyze modules within MPA. These contributions will be presented in Sections 5.3 and 5.2, and led to the publications [ALM10, AM10a].
- We identified and defined formally the *causality problem* in Real-Time Calculus. This problem occurs when the upper and lower curves of a pair of arrival curve provide conflicting constraints (e.g., specify a stream that must emit at least 4 events, but at most 3 in a given time window). We explain why the MPA framework did not have to deal with the causality problem with traditional pure-RTC analysis, and how it is problematic when connecting to some other formalisms. We provide a transformation called the *causality closure* that allows getting rid of these conflicting constraints. The causality closure is first defined for the most general case, and then issues specific to some particular classes of curves are solved. This contribution will be presented in Section 5.4.1 and has been published in [AM10b, AM11].

1.3 Organization of the document

The document is organized in two parts. The first part deals with functional properties and their verification in several contexts: we first present verification techniques for general-purpose programs in Chapter 2, and present some uses of verification tools and new compilation techniques in the context of SystemC/TLM in Chapter 3. The second part presents some work on non-functional properties. Chapter 4 deals with time, power consumption, and temperature of systems-on-chips modeled at the TLM level. Chapter 5 presents our contributions on modular performance analysis with real-time calculus, which are very abstract models used to derive timing properties.

Part I

Functional Properties

Chapter 2

Verification and Abstract Interpretation

Any sufficiently advanced bug is indistinguishable from a feature.

— Rich Kulawiec

This chapter presents the work on verification of general-purpose languages. They range from simplistic tools that only reproduce some well-known verification methods to new techniques, combining or improving recent advances in static analysis. One point they have in common is that all the tools deal with common programming languages, as opposed to formal languages specifically designed to be verified. In practice, some of the tools do have limitations and manage only a subset of the language being studied, but the subset is always large enough to verify at least small programs, without any human intervention and manual translation, following Gérard Berry's *WYPIWYE* (What You Prove Is What You Execute) principle [Ber89].

The next section starts with some simple experiments to implement traditional verification techniques in some particular contexts (as part of short internships). The reader familiar with abstract interpretation and model-checking may want to skip to Section 2.3.2 where recent advances in abstract interpretation are presented. Finally, Section 2.4 present new techniques that combine and improve state-of-the-art abstract interpretation.

2.1 Abstract Interpretation of Java Sequential Programs

Our first goal was to apply static analysis on Java programs, which was done during the internship of Loic Crétin, co-supervised by David Monniaux and myself. One nice property of Java in this context is that it is a "safe" language, in the sense that "segmentation faults" and most undefined behavior that are common in C are avoided by construction in Java (forced variable initialization, no pointer arithmetic...). Dealing with a general purpose language however has a cost, since writing a Java compiler front-end from scratch alone would have been longer than the time allocated to the internship. It was therefore clear that we had to maximize code reuse, and most technical constraints derived from this. This emphasis for code reuse can be seen as a technical constraint, but it also has an interesting side effect: it forced us to focus on the fundamental concepts of abstract interpretation, and delegate any technical detail to other tools.

We clearly needed a front-end that could make an intermediate representation easily available. We have chosen to reuse the Java front-end of the Eclipse IDE, by implementing the analyzer as an Eclipse plugin: refactoring tools and smart completion rely on the fact that the IDE permanently has access to the Abstract Syntax Tree (AST) of the program being edited, and this AST is made available to plugins.

Also, we obviously didn't want to re-implement a non-trivial abstract domain. The Apron library [JM09] provides a nice abstraction layer on top of several abstract domains. The caller of the library only uses abstract operators, and the library decides on the abstract domain implementation. Since Apron is written in C, this forced us to use a bridge between Java and C, for which we chose Java Native Access (JNA).

On the theoretical point of view, one (unusual) choice was to work on the abstract syntax tree of the program directly, instead of working on a control-flow graph as most abstract interpretation tools do. This choice was partly inspired by the implementation of ASTREE [CCF+05] analyzer. One obvious advantage is that the AST was already available, and using it directly avoided the need for another intermediate format. The approach has more subtle advantages: it makes the iteration over the program a very simple recursive tree traversal (while the order in which control points should be visited in control-flow based abstract interpretation is a non-trivial issue), and can be more efficient in terms of memory, since the abstract values associated with program points can be stored in local variables, and freed when the analysis of a part of the program is completed. On the cons side, working on the AST forces a well-structured program: `goto` statements would be hard to deal with, but this is not a problem in Java where this construct does not exist.

Figure 2.1 shows an example program with its abstract syntax tree. We use `?` to represent non-determinism (i.e., "`y = ?`" can yield any value for `y`). We explain the behavior using the intervals abstract domain. Clearly, while the analysis will not discover any property on the value of `y`, it does discover $\texttt{x} \geq 0$ at the end of the program (we work with mathematical, unbounded integers, although the actual semantics would be modulo 2^n in Java).

```
x = 0;
y = ?;
if (y > 0) {
    x = y;
} else {
    x = -y;
}
```

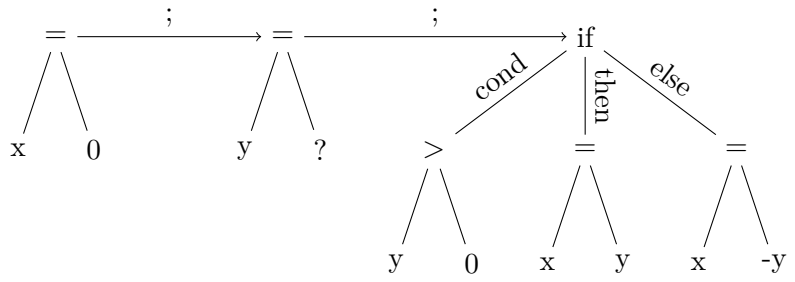


Figure 2.1: Example Program and Abstract Syntax Tree

The management of sequence (`;` in the code and AST) is done the usual way. The `if` statement is analyzed using Algorithm 1.

Algorithm 1 Analysis of `if` statements

function ANALYZE(`IfStatement tree`, $I_{precondition}$)
  $I_{cond} \leftarrow$ ANALYZE(`tree.cond`)
  $I_{then} \leftarrow$ ANALYZE(`tree.then`, $I_{precondition} \sqcap I_{cond}$)
  $I_{else} \leftarrow$ ANALYZE(`tree.else`, $I_{precondition} \sqcap \text{not}(I_{cond})$)
  return $I_{then} \sqcup I_{else}$
end function

Applied to our example, this computes $I_{cond} = y > 0$, $I_{then} = x \in]0, +\infty[\sqcap]-\infty, +\infty[) = x \in]0, +\infty[$ and $I_{else} = x \in ([0, +\infty[\sqcap]-\infty, +\infty[) = x \in [0, +\infty[$. It then derives the invariant $I = I_{then} \sqcup I_{else} = x \in [0, +\infty[$ at the end of program. Note that the recursive calls to ANALYZE could be done on any tree, including nested `if`, `while`, ... (in the actual code, this uses a visitor design pattern, so there is one function ANALYZE per kind of node).

The management of loops requires a bit more work, but remains relatively simple too. The

pseudo-code is given in Algorithm 2:

Algorithm 2 Analysis of `while` loop

function ANALYZE(`WhileStatement tree`, $I_{precondition}$)
  $I \leftarrow I_{precondition}$
  repeat
    $I_{old} \leftarrow I$
    $I_{cond} \leftarrow$ ANALYZE(`tree.cond`, I)
    $I \leftarrow I \nabla$ ANALYZE(`tree.body`, $I \sqcap I_{cond}$)
  until $I = I_{old}$
  return I
end function

One can see the fixpoint iteration, and the use of the *widening* operator ∇ instead of an abstract union $\sqcup$ to enforce convergence. The widening operator satisfies by definition $x \sqsubseteq x \nabla y$, $y \sqsubseteq x \nabla y$ and for any sequence x_n, the sequence y_n defined by $y_0 = x_0$ and $y_{n+1} = y_n \nabla x_{n+1}$ converges in a finite number of steps.

Interestingly, with this implementation, the wording "abstract interpretation" is really meaningful since the implementation is very similar to the one of a *concrete* interpreter, but manipulating abstract values.

This small tool was essentially an exercise in style, but it was interesting to show that a static analyzer for a reasonable subset of a real-language could be developed in a few hundred lines of code. Small variants of the analysis algorithm can be implemented quickly too, for example, implementing a narrowing iteration to increase the precision basically consists in adding one more application of the transition function before the **return** statement of Algorithm 2.

One unfortunate lesson to learn from this experience though is that code reuse can also lead to many technical complications. Depending both on Eclipse's plugin API and on C code was probably a mistake as it brought us the technical issues of multi-language programming, and we probably spent more time fighting against bugs of the tools we used than on real theory.

2.2 Model-checking Concurrent Java Programs

The year after, we offered a similar internship to Romain Salles. The goals were similar to the one of Loic Crétin, in that it was essentially an exercise to understand the basics of formal verification. Learning from our past experience, we avoided heavyweight dependencies, to avoid too much distraction due to purely technical issues. On the other hand, we kept the idea of working on a general purpose language and the choice of Java. This time, we experimented model-checking of concurrent Java programs.

Clearly, writing a new model-checker was out of reach given the time limitations, so we opted for a translation-based approach: compile the program to be verified into the input language of an existing verification tool. In this case, the verification tool was *NuSMV* [CCG+02], a tool inspired from *SMV* [McM93] offering several symbolic model-checking algorithms.

Again, we didn't want to write our own compiler front-end. We decided to verify the Java byte-code instead of working on the source code (or an abstract representation like an AST). First, verifying the byte-code can give more guarantees because it gives sound results even in the presence of compiler bugs. Also, working on byte-code is sometimes easier since syntactic sugar in the source code has already been lowered to simpler constructs. The *ObjectWeb ASM* Library [BLC02] allowed us to easily read and manipulate Java bytecode.

The choice of bytecode as input language however had a drawback: Java's bytecode is stack-based, and we cannot encode an unbounded stack in NuSMV. Recovering the expression abstract

tree from the bytecode would be possible but non-trivial, so we chose the brute-force approach, by computing statically a bound on the size of the stack (which is easy since we do only intra-prodecural analysis), and encode the stack as a static array of variables (in our case, an array of Boolean variables since we dealt only with Booleans).

The tool was able to model-check Dekker's mutual exclusion algorithm. It is clearly limited in the accepted subset of Java as input language, and also makes a naive approximation of the underlying memory model: the program makes the assumption that memory accesses are all atomic and immediately visible to other threads. The Java memory model is indeed more relaxed (operations are not atomic, and they are guaranteed to be visible by others only after an explicit synchronization).

2.3 From Bounded Model-Checking to Abstract Interpretation

2.3.1 Bounded Model-Checking

Continuing on the series, we offered an internship on the development of a small model-checker based on SAT-solving (Julien Henry's Master I). We considered only loop-free functions, and intra-procedural analysis, so the problem was decidable by construction (i.e., we did not have to bound the size of execution explicitly). We used *LLVM* [LA04] as a compiler infrastructure, which allowed us to take C, C++ or even Ada as input language: the LLVM front-ends (clang or llvm-gcc) could be used as independent, external tools, to generate the LLVM bitcode. The LLVM *bitcode* is the intermediate format, available in either human-readable form, binary file, or data structure.

It is in *Static Single Assignment (SSA)* [CFR+91] form. This means that a succession of assignments on the same source variable like the one in Figure 2.2.(a) is translated into a set of assignments on multiple versions of the variable, as is done in Figure 2.2.(b).

```
x = 0;
x = 42;
x = x + 1;
```

(a) Source program

```
x0 = 0;
x1 = 42;
x2 = x1 + 1;
```

(b) Program in SSA form

Figure 2.2: Static Single Assignment (SSA) example

This makes the encoding into a SAT or SMT formula straightforward for most constructs: each SSA variable takes only one value during an execution of the code, so the translation into SMT problem can basically create one SMT variable per SSA variable.

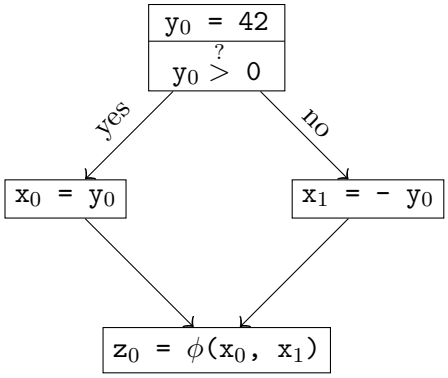
The `if` statements require some attention. The encoding into SSA already converted conditionals into a *Control-Flow Graph (CFG)* as illustrated by Figure 2.3. Control-flow splits are done by a conditional jump at the end of a basic-block. Control-flow joins use the special ϕ instruction to get the value from the right incoming branch. $\phi(x_0, x_1)$ reads simply as "if the control comes from the first incoming branch, then return x_0 else return x_1".

The encoding as an SMT problem uses one SMT Boolean variable b_i per basic block. b_i is true if the path considered goes through the basic block i. Then, ϕ statements can be encoded as `if` statements over b_i variables. For example, the program in Figure 2.3 is encoded in the logic formulas of Figure 2.4.

Loops (or generally, any edge going backward in the control flow) cannot be directly encoded with this scheme. A loop unrolling could have been used (like CBMC [CKL04]) to perform bounded-model checking, that is, prove that no property violation can occur in less than N loop iterations for a given N. This wasn't done by lack of time, and because the encoding could be

```
y = 42;
if (y > 0) {
    x = y;
} else {
    x = -y;
}
z = x;
```

(a) Source program



(b) Program in SSA form

Figure 2.3: SSA and Conditional Statements

$$
\begin{aligned}
& y_0 = 42 && \wedge\, b_0 = \text{true} \\
\wedge\, & x_0 = y_0 && \wedge\, b_1 = y_0 > 42 \\
\wedge\, & x_1 = -y_0 && \wedge\, b_2 = y_0 \leq 42 \\
\wedge\, & z_0 = \text{if } b_1 \text{ then } x_0 \text{ else } x_1 && \wedge\, b_3 = b_1 \text{ or } b_2
\end{aligned}
$$

Figure 2.4: Encoding of the Program as SMT formula

used as a building block of an abstract interpreter without having to deal with loops, as we will show in the next section.

2.3.2 State of the Art Abstract Interpretation

In parallel with the work on bounded-model checking, we worked with Marc Pégon on abstract interpretation, also using LLVM. The result was a tool implemented using a modern framework able to deal with a reasonable subset of C, but using only traditional abstract interpretation techniques. Then, the work continued with Julien Henry's Master II, in which we caught up with state-of-the-art abstract interpretation techniques, still using LLVM. In addition to the traditional abstract interpretation, two techniques were implemented in a tool called *PAGAI*: *guided static analysis* and *path focusing*.

```
1  void rate_limiter() { // (s)
2      int x_old = 0;
3      while (1) { // (l)
4          // non-deterministic number between -100000 and 100000
5          int x = input(-100000, 100000);
6          if (x > x_old+10) x = x_old + 10;
7          // (m)
8          if (x < x_old-10) x = x_old - 10;
9          x_old = x;
10         // do nothing any number of times
11         while (wait()) {} // (w)
12     }
13 }
```

Figure 2.5: Motivating example: a rate limiter in C

As a motivating example, consider the C program in Figure 2.5, implementing a rate-limiter.

This program has several specificities that make it hard to analyze with traditional abstract interpretation:

1. It contains two related `if` statements in sequence. A naive analysis will have to perform an abstract union at the control-flow merge (labeled *(m)*) in between. As a result, the values obtained from the first "then" branch are used in the analysis of the second "then" branch, even though the path where both `if` statements enter the "then" branch is infeasible.

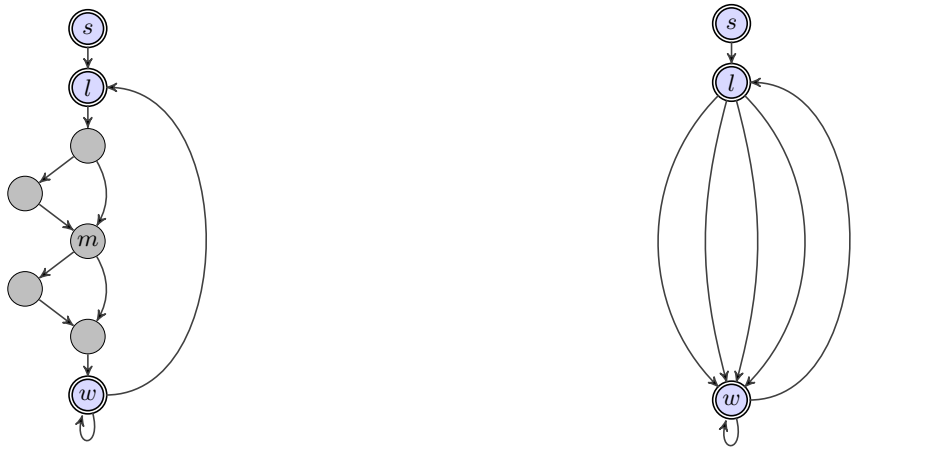
2. It contains a nested `while` loop (line 11), which makes the narrowing inefficient.

Guided Static Analysis

Guided static analysis [GR07] performs the traditional ascending and narrowing iterations on a subset of the CFG, and then considers larger subsets. The analysis terminates when the subgraph cannot grow anymore. The narrowing iterations are applied at each intermediate steps of the analysis, before considering a new subgraph. It can recover precision that would have been impossible to recover at the end of the analysis. In the example above, issue 2 makes narrowing ineffective. Guided static analysis alone is not sufficient in this case, but the combined technique presented below solves the problem (by never including this transition in the subgraph, since it is not needed to make the invariant grow). Typically, a widening operator can activate infeasible transitions, but guided static analysis will not consider such transition before applying one narrowing sequence. Guided static analysis avoids propagating the precision loss of the widening operator to the whole CFG by applying it on a subset.

Path Focusing

Path focusing [MG11] can improve precision by avoiding unnecessary abstract unions operations at control flow merges (issue 1 above). In traditional abstract interpretation, abstract unions are computed after each if/then/else construct, resulting in loss of precision if the union of abstract values cannot be represented exactly by an abstract value (e.g., if the union is not convex when working with polyhedra, a convex hull has to be computed). Instead of working on the control flow graph (CFG), path focusing works on the expanded *multigraph*: it considers only a subset of control points P_R (typically, loop heads), and paths from one point of P_R to another as edges of the multigraph. The expanded multigraph of the rate-limiter is shown in Figure 2.6.



(a) Control Flow Graph (b) Expanded multigraph

Figure 2.6: CFG and multigraph for the rate-limiter with $P_R = \{s, l, w\}$

The multigraph can be exponentially larger than the source program, but is never represented explicitly. Instead, it is modeled as an SMT formula that we call the *implicit multigraph*. The algorithm queries the SMT solver "is there a path starting from the current invariant candidate that would make the invariant grow?". In other words, it asks for the existence of a path p that starts from the invariant candidate, complies with the semantics of the program, and ends outside the invariant candidate. If the query is satisfiable, the SMT solver provides a model that can be interpreted as a path between P_R points, and the corresponding abstract transformation is applied on this path. If the query is unsatisfiable, then the analysis has already reached the fixpoint, and the invariant candidate is actually an invariant.

2.4 New Abstract Interpretation Techniques

At the start of Julien Henry's Ph.D, the tool PAGAI was implementing state-of-the-art abstract interpretation techniques, including guided static analysis and path focusing that were never experimented on real programs before, but included no new techniques. Experiments showed that path focusing and guided static analysis could both provide tighter invariants, but in different cases. PAGAI was then improved to combine both techniques and to compute disjunctive invariants, i.e., compute the invariant as a list of abstract values at each program points (which can avoid abstract unions that would lose information) [HMM12b].

2.4.1 Combining Path focusing and Guided Static Analysis

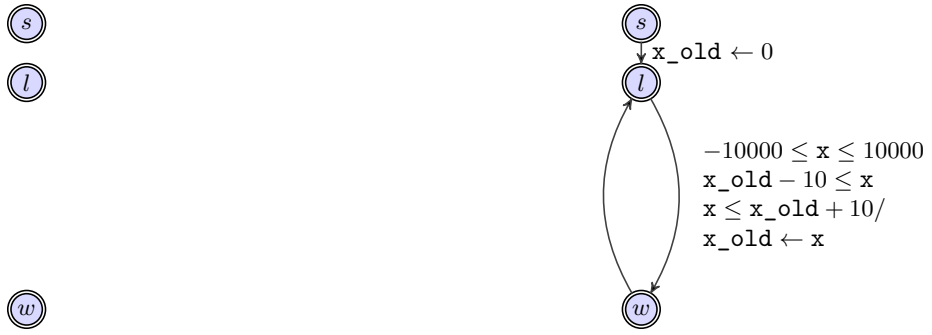
The combined algorithm of PAGAI uses SMT queries to perform the analysis on the implicit multigraph (like path focusing), but works on an ascending sequence of multigraphs like guided static analysis. An overall view of the algorithm can be found in Algorithm 3.

Algorithm 3 Overall view of the combined algorithm

```
1:  A' ← initial locations   ▷ Points in P_R than may be the starting point of new feasible paths
2:  A ← ∅                    ▷ Points in P_R that may be the source of an interesting path
3:  P ← ∅                    ▷ Paths in the current subset
4:  while A' ≠ ∅ do
5:      repeat               ▷ Compute a larger sub-multigraph
6:          Compute new path starting from A'   ▷ Updates A, A' and P
7:      until A' = ∅
8:      repeat               ▷ Ascending iterations on sub-multigraph P
9:          Get and remove p_i from A
10:         for all path (p_i → p_j) that can make the p_j invariant grow do
11:             Update invariant of p_j
12:             Add p_j to A and A'
13:         end for
14:     until A = ∅
15:     Perform narrowing iterations on P
16: end while
```

The outer loop iterates over the ascending sequence of multigraphs. Like guided static analysis, the algorithm makes it possible to run a narrowing sequence on a sub-multigraph before analyzing new paths. The inner loop lines 8-14 essentially applies path focusing to the sub-multigraph P.

On the example of the rate-limiter of Figure 2.5, the analysis would compute the ascending



(a) Step 1 (b) Step 2

Figure 2.7: Ascending sequence of sub-multigraphs

sequence of sub-multigraphs of Figure 2.7: initially, the multigraph is empty ($P = \emptyset$, Figure 2.7.(a)). Then, a set of 3 "interesting paths" are computed (Figure 2.7.(b)).

"interesting paths" are paths that are not yet in P, and that can make the candidate invariant grow with a single iteration. They are computed using SMT solving. Initially, the only interesting path is $(s \to l)$, which yields the candidate invariant `x_old = 0` at program location l. Then, there are several interesting paths from l to w, and the SMT solver may return any of them. If it returns the path going through both "else" branches of the `if` statements, then the candidate invariant at w will be $\texttt{x\_old} \in [-10, 10]$. The self-loop around w is not considered as it does not make the invariant grow. The back edge from w to l makes the invariant grow at l and is therefore added to the sub-multigraph. Other transitions from l to w are not considered as they would not make the invariant grow.

Then, path focusing ascending iterations are computed on the sub-multigraph, and narrowing is applied to recover precision. Since the self-loop is not considered, the narrowing can recover $\texttt{x\_old} \in [-10000, 10000]$ at points l and w. No other interesting path are found afterwards so the analysis terminates.

2.4.2 Disjunctive Invariants

Another contribution of PAGAI is the ability to compute *disjunctive invariants*. The traditional abstract interpretation algorithm computes one invariant at each program point. When two candidate invariants are computed for the same point, an abstract union is performed to merge them into one. Instead, we keep the union symbolic by computing a list of *disjuncts*: each program point p_i is associated with m disjuncts $X_{i,j}$ for $j \in \{1, \ldots, m\}$, representing the invariant $\bigvee_{j\in\{1,\ldots,m\}} X_{i,j}$. An algorithm had been proposed earlier [GZ10], but relied on an exhaustive enumeration of path. PAGAI improves the existing algorithm by using SMT queries instead of an enumeration.

The idea shared between [GZ10] and the algorithm in PAGAI [HMM12b] is to store a list of disjuncts for each program points. The number of elements in the list is bounded by a fixed number M, chosen a priori. Instead of considering paths from a program location to another, the analysis considers paths from a disjunct $X_{i,j}$ to another. When considering path number k starting from $X_{i,j}$, with destination state $p_{i'}$, the value computed along the path is merged in the disjunct $X_{i',\sigma_i(j,k)}$. In other words, the mapping function σ returns the target disjunct into which the new abstract value should be merged. The σ function can be computed on the fly during the analysis: the analysis starts with an undefined σ. When the value $\sigma_i(j, k)$ is required but undefined, the analysis computes the image of $X_{i,j}$ by the path, and tries to find a target disjunct where the abstract union would not lose information. If no such disjunct is found, then we set $\sigma_i(j, k)$ to either a fresh disjunct (if less than M disjuncts have already been allocated),

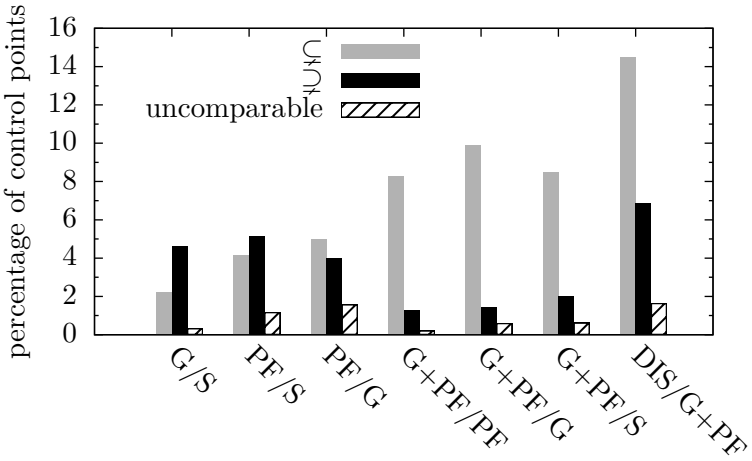


Figure 2.8: Experimental Results of PAGAI

or to the m-th disjunct.

Interestingly, the idea of using SMT queries to find interesting paths also applies to disjunctive invariant computations. Not only the choice of the path to consider can use SMT, but the computation of σ, and the choice of the source disjunct to be used can also benefit from it. The formulas submitted to the SMT solver are a bit more complex than the ones used for path focusing and the combined algorithm of Section 2.4.1, as they have to quantify also on the source and target disjunct ("is there a source disjunct for which one of the target disjuncts would grow?" instead of "is there path for which the target invariant would grow?"). Clearly, the analysis is slower, but experimental results show that the precision is improved in many cases [HMM12a].

2.4.3 Real-Life Experimentations

PAGAI being implemented as an analyzer for LLVM bitcode, it can take any program accepted by the LLVM front-end. PAGAI has no model of the actual memory, and considers only LLVM registers (essentially, local variables and no pointers). However, memory accesses are abstracted in a sound way: `store` operations are ignored, and `load` operations return a non-deterministic value. The analysis on arbitrary code is therefore sound with only a few exceptions (integers and floating points variables are considered as mathematical integers and reals respectively).

This allowed us to compare several techniques with the same tool, on real programs. Experimental comparison is necessary because all of the analysis are non-monotonic: being more precise at some point in the analysis can lead to loss of precision later (e.g., by triggering a widening), hence it is impossible for an algorithm to be strictly better than another. A summary of the comparison between techniques is shown in Figure 2.8. Techniques are classical abstract interpretation (S), *Guided Static Analysis* (G), *Path-focusing* technique (PF), our combined technique (G+PF), and its version with disjunctive invariants (DIS). The $\subsetneq$ bars (resp. $\supsetneq$) gives the percentage of invariants stronger (more precise; smaller with respect to inclusion) with the left-side (resp. right-side) technique, and "uncomparable" gives the percentage of invariants that are uncomparable, i.e neither greater nor smaller; the code points where both invariants are equal make up the remaining percentage. For example, the entry "G+PF/G" shows that the combined analysis (G+PF) is better than guided static analysis alone in almost 10% of the cases (grey bar), worse in a bit more that 1% of cases (dark bar), and uncomparable in 0.6% of cases (hatched bar). The full comparison is available in [HMM12a].

The results show that state-of-the art techniques like guided static analysis and path focusing,

and the new proposed techniques can improve the precision in a few percent of the cases, but unsurprisingly, can also lose precision.

PAGAI is open-source and available from `http://pagai.forge.imag.fr/`. The real-life experiments showed the robustness of PAGAI: the tool can actually handle large and complex codebases. It has also been used by external teams to experiment new features like localized widening [AS13].

Chapter 3

Compilation and Verification for SystemC

> *Even perfect program verification can only establish that a program meets its specification. Much of the essence of building a program is in fact the debugging of the specification.*
>
> — Fred Brooks, *No Silver Bullet (1986)*

Chapter 2 presented analysis techniques for general purpose imperative languages, with a focus on sequential programs. The following chapter will consider techniques that apply in the particular case of SystemC. Most of the work presented in this chapter was done as part of our industrial partnership with STMicroelectronics: first, during my Ph.D, and then during the OpenTLM collaborative project.

An obvious approach to analyze SystemC programs would be to consider them as any C++ code. Indeed, SystemC was built as a library, not as a new language, hence any tool that applies to C++ can, in theory, accept SystemC too. This is a very important property of SystemC, since this is what allows reusing the whole C++ ecosystem to develop and debug (e.g., GCC, GDB, Eclipse, ...). However, using C++ static analyzers to prove properties on SystemC programs would be either incorrect or inefficient, and most likely both:

- The SystemC library includes a simulation kernel that contains a scheduler. Calling SystemC primitives like `wait` can result in context-switches, implemented using low-level primitives (either POSIX thread or assembly code in the QuickThread library). Analyzing the source code of the SystemC library would therefore be very hard.
- The SystemC library is deterministic. Even if an analyzer can prove a property on a program containing user code plus the SystemC library, the proof would hold only for this particular implementation of SystemC, and would not guarantee that another valid scheduling choice would not violate the property.
- When a component sends data to another component, finding the target component requires some knowledge about the architecture of the platform, which is built at runtime in SystemC. A general purpose analyzer would require a program-wide precise pointer analysis before being able to say anything about even simple statements like `port.read()`, which is not realistic without any knowledge of the SystemC semantics.

Dedicated tools, with some built-in knowledge about the SystemC semantics, are therefore needed to apply formal methods to SystemC. The first necessary step for most SystemC aware

tools is to write a *front-end*, that is, a tool able to read a SystemC program and expose it to the back-end as an abstract data structure.

This chapter first presents a new front-end called *PinaVM* (Section 3.1), which can extract an abstract data structure containing both the behavior and the architecture of a SystemC program. PinaVM reuses some of the ideas of the existing SystemC front-end *Pinapa*, which is recalled briefly here. Then, in Section 3.2, we present approaches for formal verification of SystemC, some of which use PinaVM as a front-end. Section 3.3 presents another application of PinaVM for optimized compilation.

3.1 SystemC Front-ends

3.1.1 Challenges in Writing SystemC Front-ends

A SystemC front-end is different from a traditional compiler front-end. KaSCPar [Des06] or ParSyC [FGC+04] use traditional compilation techniques. They consider SystemC as a language with its own grammar and parse it respectively by JavaCC and PCCTS. However, such approaches are fundamentally limited by the way the platform to simulate is built in SystemC. The architecture of the platform is not directly described in the program, but instead, a part of the program called the *elaboration phase* (the body of the `sc_main` function, before launching `sc_start`) builds this architecture.

For example, Figure 3.1 shows the elaboration phase for the platform presented in Figure 1.1 page 14. In this case, the code is simple and regular enough to be parsed by pattern-matching constructs within `sc_main`. Now, consider two variants of this platform:

- An excerpt from a multi-core variant of the platform is shown in Figure 3.2. The instantiation of the CPU and the interrupt controller is now done within a `for` loop, indexed by the number of CPUs to instantiate. In this case, traditional techniques do not apply directly anymore (for example, a very naive approach would see only one `new Intc()` and consider there is only one instance of the `Intc` module). A solution would be to unroll the loop, to fall-back to simple and regular code, but loop-enrolling is not always possible.

- Figure 3.3 shows an example code to make the platform customizable. The use-case here is to allow disabling the VGA GUI display in simulation with an environment variable. In this case, there is no single platform architecture, and several runs of the same program can yield different architectures. The architecture is really *dynamic* with respect to traditional compilation techniques, but we still want SystemC-aware tools to consider it as *static*, because it will not change during simulation.

Note that both cases are real examples taken from the experiments done with the `sc-during` library (described below in Section 4.1.5). More examples of tricky cases and possible approaches and tools to deal with them can be found in [MMK10].

3.1.2 Existing Approaches for SystemC Front-Ends

Existing SystemC front-ends can be classified in several categories, depending on how they retrieve information:

Static Approaches consider SystemC as a language, and parse it with traditional compilation techniques (e.g., lex+yacc or some variants of it). They are fundamentally limited as they cannot deal with complex code in the elaboration phase as presented above.

```
int sc_main(int , char**)
{
    MBWrapper cpu("MBWrapper");
    Memory inst_ram("inst_ram", 0x00002000);
    Memory data_ram("data_ram", 0x00100000);
    Bus bus("bus");
    Timer timer("timer", sc_core::sc_time(20, sc_core::SC_NS));
    Vga vga("vga");
    Intc intc("intc");
    Gpio gpio("gpio");

    sc_core::sc_signal<bool> timer_irq("timer_irq"),
                             vga_irq("vga_irq"),
                             cpu_irq("cpu_irq");

    // initiators
    cpu.socket.bind(bus.target);
    vga.initiator(bus.target);

    // targets
    bus.initiator(data_ram.target);
    bus.initiator(inst_ram.target);
    bus.initiator(vga.target);
    bus.initiator(timer.target);
    bus.initiator(gpio.target);

    // interrupts
    vga.irq(vga_irq);
    timer.irq(timer_irq);
    intc.in0(vga_irq);
    intc.in1(timer_irq);
    intc.out(cpu_irq);
    cpu.irq(cpu_irq);

    //      port              start addr          size
    bus.map(inst_ram.target, INST_RAM_BASEADDR, INST_RAM_SIZE);
    bus.map(data_ram.target, SRAM_BASEADDR,     SRAM_SIZE);
    bus.map(vga.target,      VGA_BASEADDR,      VGA_SIZE);
    bus.map(gpio.target,     GPIO_BASEADDR,     GPIO_SIZE);
    bus.map(timer.target,    TIMER_BASEADDR,    TIMER_SIZE);

    // start the simulation
    sc_core::sc_start();
    return 0;
}
```

Figure 3.1: Elaboration phase of the platform in Figure 1.1

```
for (int i = 1; i <= NB_SLAVE; i++) {
        Intc *ic = new Intc();
        MBWrapper *cpu_slave = new MBWrapper(i);
        sc_core::sc_signal<bool, SC_MANY_WRITERS> *s =
                new sc_core::sc_signal<bool, SC_MANY_WRITERS>();
        cpu_slave->irq(*s);
        ic->out(*s);
        intc_slaves[i-1] = ic;
        cpu_slaves[i-1] = cpu_slave;
}
```

Figure 3.2: Usage of `for` loops within the elaboration

```
const char *vga_env = getenv("SC_HEADLESS");
if (vga_env == NULL || !strcmp(vga_env, "")) {
        vga_ptr = new Vga("vga");
} else {
        vga_ptr = new HeadlessVga("vga");
}
```

Figure 3.3: Configuration dependant on the command-line invocation

Dynamic Approaches only execute the SystemC program, and extract information at runtime. Quiny [SN07] is probably the only front-end in this category, using operator overloading extensively. While very elegant, it is a very limited approach.

Hybrid Approaches execute the elaboration phase to retrieve the architecture, and combine this dynamic information with static information produced by a compiler.

Both Pinapa and PinaVM fall in the last category, as we believe this is the less limited one. Both tools execute the elaboration phase and explore the data structures of SystemC to obtain the architecture. Pinapa uses GCC's AST, and PinaVM uses LLVM's bitcode for the behavior information. Other tools using the hybrid approach includes LENSE [SWD13], which relies only on the debug information for the static information. This approach is lighter as it does not require the use of a specific compiler within the SystemC front-end, but it does not extract the behavioral information (i.e., the body of functions as an AST or bitcode). It is well suited for detailed visualization, but unsuitable for formal verification. Another notable example of hybrid approach using GDB is *SHaBE* [BvL11], which uses GDB breakpoints instead of hooking into the SystemC library. Although very different in terms of implementation, SHaBE is very similar to Pinapa and PinaVM conceptually.

3.1.3 Pinapa: a First Attempt at a Dynamic Approach

Instead of using traditional compilation techniques, we propose using a hybrid approach: the elaboration phase is compiled and executed, and a traditional front-end is used to analyze the body of processes. This approach was first introduced in the tool *Pinapa* [MMMC05b], which was the front-end of the tool LusSy [MMMC05a, MMMC06] developed as part of my Ph.D.

Interestingly, Pinapa was originally written to *avoid* having to write a SystemC front-end: the goal of the Ph.D was to write a formal verification tool, and one of the motivation to choose a hybrid approach was to minimize the required effort in the front-end, to focus on the back-end.

Beside the fundamental choice for a hybrid approach, an important technical choice is to reuse an existing C++ front-end (a decent C++ front-end takes hundreds of thousands of lines

of code, so writing one from scratch was not an option). At that time, the only open-source and complete C++ front-end was the one of GCC, which provided us an Abstract Syntax Tree (AST).

Getting the architecture of the platform (called *ELAB* in Pinapa) and the AST of process bodies are therefore solved problems. However, a SystemC front-end needs to do more than this: the AST and ELAB need to be linked together:

1. ELAB contains one process handle for each SystemC process created in the platform. This process handle is normally used in simulation, and contains a pointer to the *compiled* version of the process body (the scheduler will launch the process by dereferencing the pointer). It needs to be enriched with a pointer to the *AST* for the same process.

2. The AST contains various SystemC constructs, like `port.write(value)`. The AST contains a node representing `port` as a class field (member of the containing `SC_MODULE`). In addition to this, we need, for each process instance, a pointer to the actual corresponding object. This is needed in particular to know which signal the port is connected to (i.e., where the value is written to). In the example of Figure 3.2, the module `Intc` has one port `out`, which is a field of the class `Intc`, but each instance has one port instance connected to a different CPU.

The first problem is relatively easy to solve because the name of the class and method for each processes can be obtained at runtime. Then, the process handles and the AST can be matched based on their names. The second problem required more work, and was solved in a relatively ad-hoc way by Pinapa: in expressions like `port.write(value)`, GCC provides the offset of `port` relative to the containing object (i.e., `this`). Then, Pinapa uses pointer arithmetic to compute, for each `SC_MODULE` instance, the address of the concrete object referred to in the AST.

The two main design choices of Pinapa (hybrid approach, and reuse of a C++ front-end) allowed it to be more expressive than all the other existing front-ends at this time, with a minimal effort (less than 10,000 lines of code in total). It was successfully used by external tools like Kratos [CGM+11] and a SystemC-to-bytecode compiler targeting FPGA execution [SMV09].

However, it suffers from several limitations. On the technical side, it uses the internal API of GCC, hence porting it from a version of GCC to another requires substantial work. Also, the format used in the AST represents precisely the program as written in the source code, which makes it rather complex to manipulate (for example, there are different node types for `for`, `while` and `do {...} while` loops, while it would be desirable to lower all of them into an abstract loop construct). These limitations motivated the development of a new front-end called PinaVM, presented in the next section.

3.1.4 PinaVM: a SystemC front-end Based on an Executable Representation

The initial motivation for writing the new SystemC front-end PinaVM [MM10a, MM10b] was to extract an SSA representation of SystemC processes. Indeed, experiments [BGM+09b] had shown that starting from an SSA form was very efficient when generating code in a synchronous formalism (see Sections 3.2.1 and 3.2.3 below).

The LLVM infrastructure was chosen, as it was providing an SSA intermediate form, and had a modular design with public APIs (GCC's plugin system was not available at that time). Writing PinaVM on top of LLVM would therefore also be a technically cleaner solution than Pinapa. This was the starting point of the post-doc of Kevin Marquet. During PinaVM's development, Ranjan Ravi helped with engineering aspects like automated tests, and Bageshri Karkare experimented alternative tools as part of our review [MMK10].

However, the way Pinapa recognizes SystemC constructs in the AST (see Section 3.1.3) is specific to the AST of GCC, and could not be directly reused with LLVM's bitcode. Simple constructs like `port.write(value)` are translated into a handful of bitcode instructions, not easily pattern-matchable automatically. PinaVM therefore had to develop a new and more general technique to link the bitcode representing process body to the objects representing the architecture. It is relatively easy to find the function call corresponding to `port.write(value)` in the bitcode, but the arguments of the function being (`port` and `value`) called are possibly computed with a non-trivial piece of code. PinaVM slices these pieces of code and turn them into functions that take the value of the containing component (`this`) as argument. These functions are compiled using the *Just-In-Time (JIT)* compiler of LLVM, and executed once per component with the right value of `this`. The result is attached as decoration to the bitcode, and available for back-ends.

PinaVM allowed developing several back-ends, essentially for formal verification of SystemC (presented in the next section). PinaVM is open-source, and available from `http://pinavm.forge.imag.fr/`. It is also used in tools outside Verimag, like SCiPX [LTDDS+11] that translates SystemC into IP-XACT, and received external contributions from Chia-Wei Chang, who uses PinaVM for distributed execution of SystemC programs. The front-end can handle real SystemC code, but is not very robust (there are several known bugs that are being worked on by Guillaume Sergent). The back-end parts presented below are only at proof-of-concept stage, and work only on a limited set of examples.

3.2 Verification of SystemC/TLM

3.2.1 Overview of Verification Approaches

There are many approaches to apply formal verification to SystemC programs. As explained above, tools targeting C++ cannot prove most interesting properties. One possibility is to write a completely new tool for SystemC, and other options include translation-based approaches, in which the tool acts like a compiler from SystemC to a formal language, to use the existing tools for this language. A few representatives of both approaches are presented below.

Dedicated Checkers for SystemC Verification

One of the earliest attempt at SystemC formal verification is presented in [GD03]. The tool uses user-instrumentation instead of a dedicated parser, and performs symbolic manipulations on the set of gates and registers (the *netlist*) of the circuit to prove some bounded LTL properties. The tool is limited with respect to the accepted subset of SystemC. The solving technique used in the tool does not benefit from all the advances in e.g., SAT solving that are already implemented in other solvers.

An alternative to symbolic manipulation is *runtime verification*. *SCRV* (for SystemC Runtime Verification toolkit) [HMMCM06] implements an algorithm that explores the possible schedules for a SystemC program for bounded executions. Unlike plain simulation methods, this gives strong guarantees as the possible schedules are exhaustively explored (actually, one execution is performed for each class of equivalent schedules, using dynamic partial order reduction to reduce the state-space explosion). It is a *stateless* approach, in the sense that the tool never has to store a set of states, which allows dealing with large programs with a reasonable memory usage, but limits it to bounded executions. It does not need any front-end: the SystemC program is compiled normally and linked against a modified scheduler.

Stateful exploration is also possible with execution-based tools. TLM.Open [Hel09] explores the complete state-space by executing SystemC *transitions* (piece of code between two `wait`

statements). Assuming the state-space is bounded, the exploration is complete: the tool memorizes the program states, and compares the state obtained after executing a transition with the set of already explored states. This way, the tool notices the presence of loops (unlike stateless approaches which need to unroll them, and have to bound the execution in the presence of infinite loops). TLM.open has been successfully used on relatively large case studies. It scales better than most other tools to explore a set of possible schedules, but is obviously limited when non-determinism comes from data (explicit state exploration of a handful of integer variables would not be possible in practice).

Symbolic techniques can be combined with explicit state ones. Kratos [CMNR10, CGM+11] implements an algorithm called *ESST*, for "Explicit Scheduler, Symbolic Threads", which uses explicit-state to explore possible schedules (with optimizations like partial order reduction), and symbolic software model-checking for the analysis of sequential code within threads.

Translation-based Approaches

Translation-based approaches compile SystemC into an existing language for which a verifier already exists. In these approaches, the proof is completely delegated to an external tool.

A natural choice is to translate SystemC into a language having similar semantics, hence built-in parallelism. Still, the translation has to carefully map the SystemC concepts into the ones of the target language, in particular the cooperative nature of SystemC and its scheduling policy. Synchronous formalisms are appealing, as they are expressive and efficient tools exist for them. A translation to Lustre and SMV was proposed in the tool LusSy [Moy05b] presented in Section 3.2.1. It models the SystemC scheduler as an automaton. The translation is implemented for a large subset of SystemC, and the expressiveness of the synchronous formalism makes it relatively easy to model different SystemC constructs. However, some important high-level properties like the thread structure of the original SystemC program are lost during the translation. As a consequence, applying optimizations like partial-order reduction or symmetry reduction is almost impossible.

Another choice is to use an asynchronous formalism like *Promela* [Hol91] (the input language for the SPIN model-checker), or LOTOS (to use the CADP [GLMS11] tool chain), as proposed in [GHPS09, PS08]. The semantics of the target language is closer to the one of the source language, hence it is possible to preserve the structure of the program in the translation and let the tool apply the relevant optimizations. A difficulty is then to model the scheduler accurately, and without breaking the structure of the model. We proposed an early attempt in [TCMM07], which uses a SPIN process to model the scheduler, with disappointing results. We improved the proposal in [MMJ11] (detailed further in Section 3.2.2), which somehow inlines the scheduler code directly into the Promela processes while translating SystemC threads. More strategies using Promela as a target language are proposed and compared in [CCNR11].

For more discussion about various formalisms to encode the semantics of SystemC, and the techniques to prove properties on the result, see e.g., [MMC+08].

CheckSyC [GD05] improves the work presented in [GD03]: it uses the ParSyC front-end mentioned above to extract an AST from a SystemC program, compiles it into a bounded model-checking problem, and launches the zChaff [MMZ+01] SAT solver to check the property. CheckSyC was initially designed for low-level, RTL programs. Further works by the same authors include [GLD10], where TLM SystemC programs are translated into a sequential C programs, taking the semantics of the scheduler into account. Then, traditional software model-checking tools like CBMC [CKL04] can be used to check properties.

Most translation-based tools integrate a SystemC front-end and a code-generator together. As a result, it is hard to use the front-end of a tool with the code generator of another. Since each front-end and back-end has its own limitation, it is sometimes hard to compare tools as

they do not deal with the same language subset. A solution is to use an intermediate format, representing both behavior and architecture information, but easy to generate and parse. This has never been done for Pinapa and PinaVM by lack of time, but some proposals such as the *Intermediate Verification Language (IVL)* [LGHD13] or SHaBE's XML output could help unifying the myriad of tools manipulating SystemC programs.

LusSy: Model with Synchronous Automata and Explicit Scheduler

LusSy [MMMC05a, Moy05a, MMMC06] is the tool developed during my Ph.D. Its architecture is described in Figure 3.4, and corresponds to the typical architecture of a compiler: a front-end (Pinapa to read the SystemC program, and *Bise* to encode it into an intermediate format called *HPIOM*, for Heterogeneous Parallel Input/Output Machines), an optimizer (*Birth*), and several back-ends. This allows using a variety of tools like Lesar, Nbac, Prover plugin for SCADE, NuSMV and SMV to perform the proof.

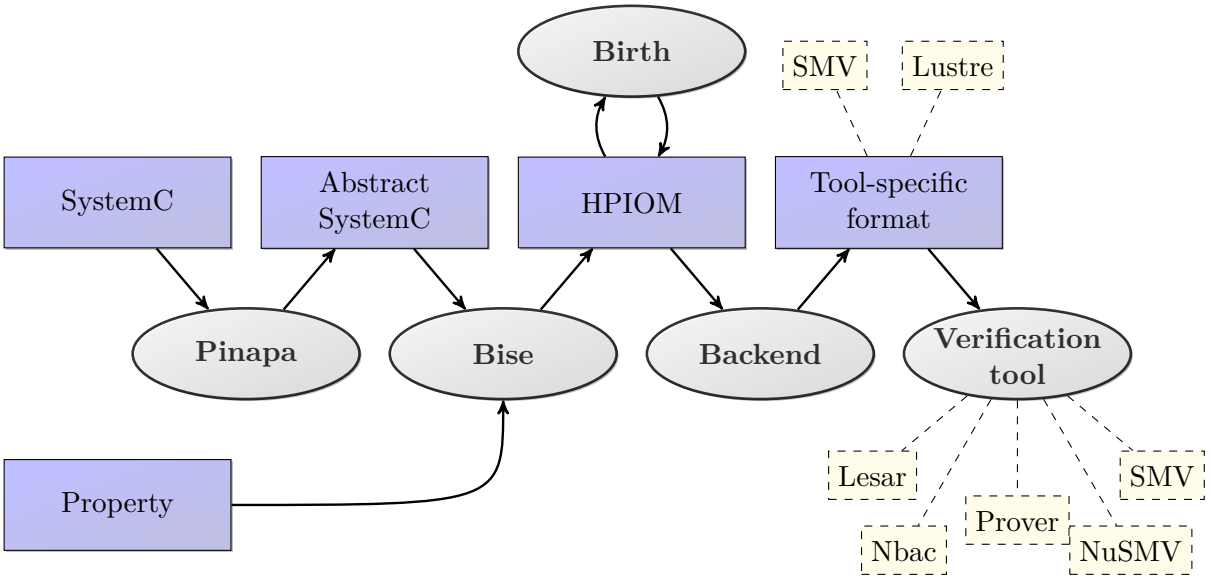


Figure 3.4: Architecture of LusSy

After reading the SystemC program with Pinapa (see Section 3.1.3 above), the component Bise extracts the semantics of the program, taking into account the property to be verified. For example, if the property to be checked is that *it is never the case that two processes write to the same* `sc_signal` *during the same* δ*-cycle*, then the semantics of `sc_signal` is modified to raise an error in such case. Bise generates the HPIOM data structure, comprising one automaton for each SystemC process, plus automata for most SystemC objects (event, signal, or TLM channel). The synchronous nature of HPIOM makes it easy to decompose a complex automaton as the product of several simpler automata. The complete scheduler, for example, consists in one automaton describing the scheduling policy, and one additional automaton per SystemC process describing its state (running, waiting for time, waiting for an event, or runnable).

The SystemC user code is compiled automatically, while the generic library code (SystemC and TLM channels) has semantics built-in Bise: we never parse nor translate these libraries, but instantiate hand-written automata that represents their semantics.

Once the HPIOM model is generated, it can be considered independently from the way it was generated. The optimizer Birth and the back-ends have no dependency on Pinapa and SystemC (in theory, other front-ends could be added although this has not been done in practice).

Originally, LusSy was generating Lustre code. The SMV back-end was added with the help

of Muzammil Shahbaz, and then extended to support NuSMV syntax by Madhav Jha. The effort to add new back-ends was relatively small, and the difficulties came essentially from unspecified behaviors of the SMV language, together with the absence of an interpreter to experimentally resolve the ambiguities (see [Moy05b], appendix B). Still, the benefits were important since SMV outperformed by far the tools dedicated to Lustre.

3.2.2 PinaVM Backend for SPIN: Asynchronous Automata Without Scheduler

An intermediate conclusion after writing LusSy was that using a synchronous, low-level formalism as an intermediate format was convenient, but removed a lot of optimization opportunities in the proof engine. An alternative is to use a richer automaton format with semantics close to the one of SystemC, such as the *micMac* automata [Cor08]. micMac automata distinguish "micro states" where context-switch is not allowed, and "macro states" which correspond to SystemC's `wait` statements. The non-preemptive nature of SystemC is modeled by the semantics of the product of two micMac automata. As no built-in tool exist for micMac, verifying a set of micMac automata means either writing a new model-checker or encoding the micMac semantics in another format.

A first attempt at the later approach is proposed in [TCMM07], where an encoding of micMac into Promela is defined. The proposed encoding is a straightforward automata encoding (using integers to represent states, one variable S_i per automata to store the current state, and modeling the transition function as a switch statement on S_i), plus a mechanism to forbid context-switches on micro-states: one global variable M stores the value of the automaton being run. The special value $M = 0$ means no automaton is running. Transitions from micro states in S_i are guarded with $M == i$ ("the process being ran is the current automaton"), and transitions from macro states are guarded with $M == 0$. Transitions to a micro state set M to the identifier of the automaton containing the transition, and transitions to macro states set $M = 0$. Somehow, the variable M can be seen as a minimalist automaton representing the SystemC scheduler.

The proposed encoding can model the SystemC semantics, but the performance is disappointing: the transitions in the state-machine seen by SPIN correspond to the ones of the micMac automaton, not to the SystemC transitions of the original model. Partial-order reduction is not possible on this model, as the variable M creates dependencies between transitions. Also, the fine granularity forces SPIN to store many intermediate states that are actually irrelevant for model-checking.

In [MMJ11], we proposed a much more efficient encoding, which maps SystemC constructs to Promela ones more directly. The SystemC scheduler is not modeled as a separate automaton, but its semantics is inlined into the automata modeling SystemC processes. For example, modeling an event notification assigns some variables to 0, while waiting for the event blocks the process while the variable is not 0 (see Figure 3.5). Time is modeled with one integer variable per process (called *deadline variables*), representing the next instant when the process will run (and the current time while running). There is no variable representing explicitly the current SystemC time, but this time can be computed as the minimum of each process deadline variable (see Figure 3.6). Combining the encoding of events and the encoding of time requires some care, as the deadline variable of a process waiting for an event is meaningless with respect to time, so the current SystemC time becomes the minimum of each processes deadline variable, *except* the ones of processes waiting for events (see Figure 3.7).

Verifying the correctness of an encoding is a hard problem. Ideally, one would want to express the encoding as a function f, transforming a SystemC program into a SPIN program, and then verify "$\forall p, f(p)$ has the same semantics as p". This requires quantifying over programs (which means model-checking is not sufficient, and the problem is clearly undecidable), and having

p::**wait**(E^k)**:**	p::E^k.**notify**()**:**
1 $W_p := k$	3 $\forall i \in P \mid W_i == K$
2 blocked($W_p == 0$)	4 $W_i := 0$

Figure 3.5: Encoding events alone

p::**wait**(d)**:**
1 $T_p := T_p + d$
2 blocked($T_p == \min_{i \in P}(T_i)$)

Figure 3.6: Encoding time alone

another formal semantics to compare with (which raises the question of the validity of the other semantics...). Instead, we wrote a set of desirable properties for an encoding, and checked them (using SPIN) on a set of test-cases for our encoding. This does not prove anything formally, but increases our confidence in the encoding. It *did* allow finding some bugs in early versions of the encoding.

Unlike [TCMM07], the encoding proposed in [MMJ11] is fully automatic, and is implemented as a PinaVM back-end. It is distributed together with PinaVM. Unsurprisingly, the performance of SPIN on models generated by the promela back-end for SPIN were orders of magnitude better than the previous ones.

The original goal was to write a PinaVM back-end for the *ConcurInterproc* [Jea09] tool. ConcurInterproc is based on abstract interpretation and can both deal with concurrency between processes and numerical variables. Function calls can either be inlined, or managed by the tool (that represents an abstraction of the call stack internally to perform inter-procedural analysis). Connecting PinaVM to ConcurInterproc would have allowed us to use abstract interpretation on concurrent programs (this was already possible using Nbac on the Lustre code generated by LusSy, but Nbac had to recover a control-structure and could not exploit the structure of the SystemC program). The connection was not possible because ConcurInterproc was not mature enough at the time PinaVM and its back-ends were written.

3.2.3 Transforming C++ and SystemC Code into Synchronous Languages using SSA

Another direction to improve the translation done by LuSsy is to keep the idea of the synchronous formalism, but to optimize the translation by minimizing the number of transitions and *state*

p::**wait(d)**:	
1 $T_p := T_p + d$	
2 blocked($T_p == \min_{\substack{ß \in P \\ W_i == 0}}(T_i)$)	
p::**wait**(E^k)**:**	p::E^k.**notify**()**:**
3 $W_i :=$ K	5 $\forall i \in P \mid W_i == k$
4 blocked($W_i == 0$)	6 $W_i := 0$
	7 $T_i := T_p$

Figure 3.7: Encoding events and time

variables. State variables are variables whose state must be stored between two transitions (i.e., variables for which a `pre`[1] statement is used in Lustre, or a `next` statement in SMV). One way to achieve this goal is to use the similarity between the SSA (Static Single Assignment) form used in compilers and synchronous languages:

SSA variables are assigned once during a basic block execution, and synchronous programs variables also have one and only one value during an instant. As a result, there is a natural mapping from SSA to synchronous languages, where one instant in the synchronous program corresponds to one, or several basic blocks (actually, a new instant is needed only for back-edges of the control flow graph). The encoding within one instant is very similar to the encoding into a SAT property presented in Section 2.3.1.

Preliminary works on the encoding of C/C++ code into the synchronous language Signal [BLGJ91] had already been proposed [KTBB06], and some comparative studies of LusSy's behavior on sequential code with the encoding into Signal showed the potential benefit of SSA [BGM+09a].

An example program and its encoding into Lustre is shown in Figure 3.8. The original scheme was presented for the Signal synchronous language, and uses clocks to encode the control flow graph. The version shown here is a single-clock variant in Lustre, using Boolean variables instead of clocks. The transformation of C into SSA creates several versions of x, but only x_1 is a state variable (as it is defined by a ϕ statement following a back-edge in the CFG). As a result, only x_1 is a state variable. Similarly, basic block 1 is the target of a back-edge, and is activated one instant after its predecessor is (hence the `pre` statement in `bb_1`'s definition), but others are activated instantaneously. For example, `bb_1`, `bb_2`, `bb_3` and `bb_4` can be true at the same time. Essentially, one synchronous instant correspond to one transition in the multigraph presented in Section 2.3.2.

However, the existing translation was proposed only for sequential code, i.e., plain C++ code as opposed to concurrent SystemC programs. In [BGM+09b], we proposed an extension of the existing scheme to support basic SystemC constructs. The translation treats SystemC constructs in a particular way:

- They generate code that "pause" for one instant: the code generated after the SystemC statement is not executed in the same cycle as the code preceding it.
- Extra signals are generated to let the SystemC process communicate with the scheduler. In particular, the code generated for the SystemC process emits one signal `end_processing` to model the end of a transition, and receives a signal `running` from the scheduler to wake it up after a `wait` statement.

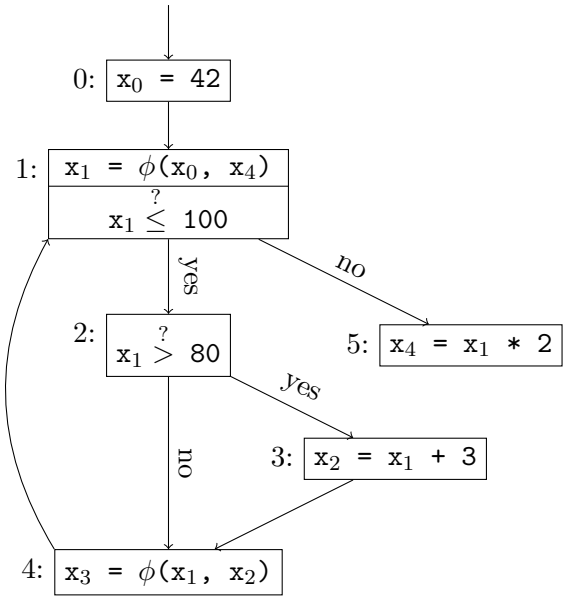
Unlike LusSy, the complete scheduler is not modeled, but the principle is laid down to model non-preemptive parallelism and a scheduling policy. The automata used in LusSy's scheduler could, in theory, be used together with the SSA-based translation.

In practice, however, the prerequisite to write a complete and automatic translations are to use an SSA-based SystemC front-end. LusSy used a SystemC front-end which is not SSA-based, and [BGM+09b] uses a plain C++ front-end (GCC), which isn't aware of SystemC constructs, and does not extract the architecture of the platform. It would be interesting to adapt the translations of [BGM+09b] and the scheduler model of LusSy to PinaVM (This was indeed one of the original motivations for writing PinaVM).

[1] The `pre` statement in Lustre means "previous". It is in general used together with the `->` ("initialization") operator . For example, `y = 0 -> pre(x)` means that the value of `y` is initially 0, and then corresponds to the value of `x` at the previous clock tick. `z = 0 -> (pre(z) + 1)` can be used to define `z` as a counter taking values 0, 1, 2,...

```
x = 42;
while (x <= 100) {
    if (x > 80) {
        x = x + 3;
    }
    x = x + 1;
}
x = 2 * x;
```

(a) Source program



(b) Program in SSA form

```
-- control-flow graph:
-- which basic block is executed?
bb_0 = true -> false; -- initially true, then false
bb_1 = false -> pre(trans_0_1 or trans_4_1);
bb_2 = trans_1_2;
bb_3 = trans_2_3;
bb_4 = trans_2_4 or trans_3_4;
bb_5 = trans_1_5;
-- which transition is taken?
trans_0_1 = bb_0;
trans_1_2 = bb_1 and     (x_1 <= 100);
trans_1_5 = bb_1 and not (x_1 <= 100);
trans_2_4 = bb_2 and not (x_1 > 80);
trans_2_3 = bb_2 and     (x_1 > 80);
trans_3_4 = bb_3;
trans_4_1 = bb_4;

-- values of variables
x_0 = 42;
x_1 = 0 -> pre(if trans_4_1 then x_4 else x_0);
x_2 = x_1 + 3;
x_3 = if trans_2_4 then x_1 else x_2;
x_4 = x_3 + 1;
```

(c) Lustre encoding

Figure 3.8: Imperative to synchronous, through SSA

3.2.4 PinaVM Backend for 42

Another experimental back-end for PinaVM was written by Pierre-Yves Delahaye. It implements the translation from SystemC/TLM to the *42 component model* [Bou10, MB07] proposed in [BMF09]. More precisely, the translation of a SystemC module generates a *control contract* for the module, which can be seen as a non-deterministic abstraction of the module in the form of a state-machine.

The purpose of the translation is to allow the compositional design of a SystemC program. Ideally, a contract should be written for each module, and then the module should be implemented in SystemC. Contracts should compose well (i.e., the guarantee of a module should imply the assumption of the modules it communicates with), and each module should comply with its contract. However, it is common when starting a new program to reuse existing modules, and these existing modules may not have a 42 contract. To allow a system-level validation of the contracts, it is necessary to extract contracts from existing SystemC modules, and the 42 backend for PinaVM allows doing this automatically for some components.

During the translation, the data part of the program is abstracted away, and the control-structure is compiled into an automaton. Only SystemC primitives (`wait`, `notify`, ...) and communication with other modules (function calls in the case of TLM communications) are taken into account in the translation.

The translation is only a proof of concept, but could be extended to extract "better" contracts from SystemC code. Clearly, the translator can only extract an abstraction of the behavior, not the intention of the programmer. A "better" contract in this case is a contract that reflects the intention of the programmer in a better way. This could be achieved by taking into account part of the data used in the program (e.g., Boolean variables that can be encoded into the control flow graph), or implement some common SystemC patterns (like `switch` statements on address within target modules, commonly used in manual modeling of register banks, or equivalent patterns for vendor-specific libraries).

3.3 Another Application of Compilation: Optimizing Compiler

3.3.1 Motivations and Principle

One major strength of SystemC is that being built as a C++ library, all the C++ tools are also available for SystemC. It is, however, also a weakness: general purpose tools for C++ do not exploit the specificities of SystemC. We already discussed the issues in the introduction of Chapter 3 in the case of verification tools, and in Section 3.1 for compiler front-ends. Similar arguments apply to the case of compiler optimizers and code generators: since the architecture is built at runtime, communication from a module to another is seen as something dynamic by the C++ compiler. This section presents a SystemC-aware compiler optimizer called *Tweto* (TLM With Elaboration-Time Optimizations), implemented as a PinaVM back-end.

Consider the case of a transaction routed by a bus, illustrated in Figure 3.9. The code within the initiator component creates a payload object and sends it to the bus using a transport method of an initiator socket (in Figure 3.9, this is wrapped in a convenience `socket.write(...)` method). The initiator socket contains a pointer to the bus, hence can forward the payload to the bus component, which does the address decoding. This can be done using a binary search over the addresses of the target components (in the example, there are 3 target modules and the address map is represented on the top right of the figure). Address decoding allows sending the payload to the right socket and finally to the right module. On overall, the complete transaction requires several virtual method calls and one search in the address space to be executed.

The cost of a transaction is particularly frustrating when the programmer already knew which component was targeted in the transaction when writing the code for the initiator module. In

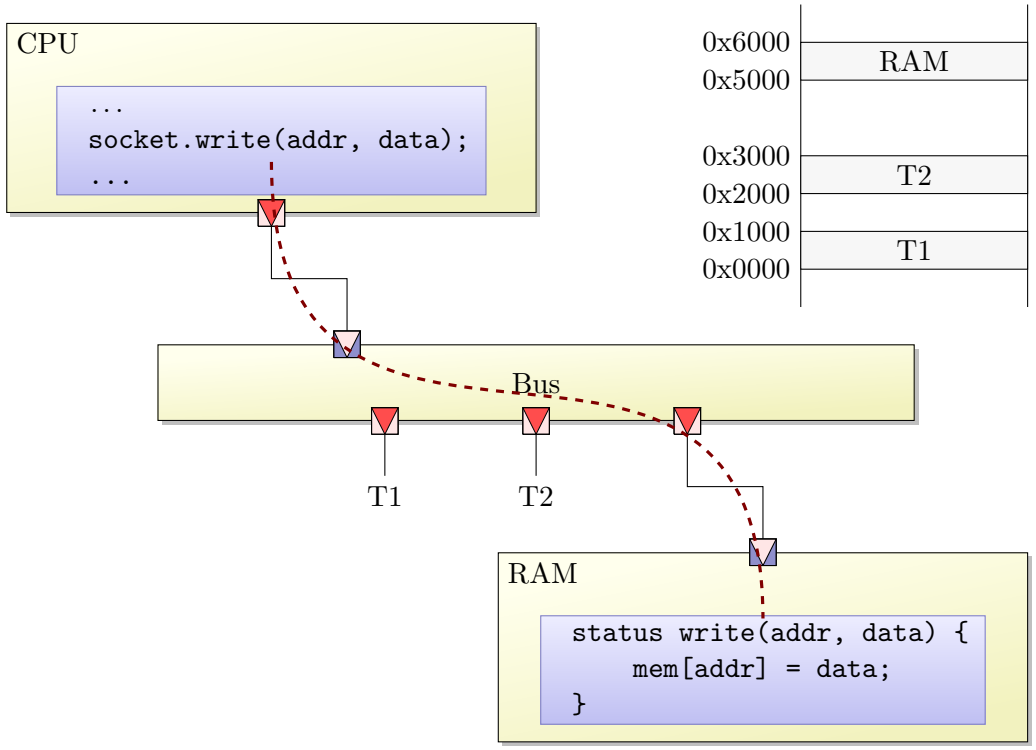


Figure 3.9: Typical Transaction Journey

practice, it is actually common to initiate a transaction with statements like

```
socket.write(VGA_BASEADDR + VGA_INTERRUPT_OFFSET, 1);
```

where `VGA_BASEADDR` and `VGA_INTERRUPT_OFFSET` are predefined constants that represent respectively the absolute address of a component (the VGA controller) and the local offset of a register (`INTERRUPT`). Another example is the model of a CPU using an instruction set simulator (ISS): the "fetch" operation will retrieve instructions from the program memory, which in most platforms always corresponds to the same target component (typically, a RAM modeled by a large array).

3.3.2 Direct Memory Interface: a Manual Optimization

The *Direct Memory Interface (DMI)* of TLM-2 was designed to avoid the transaction performance overhead, by allowing the transaction to bypass the interconnect model. Instead of sending one generic payload for each transaction, the initiator can request a direct pointer to the internal storage (as an `unsigned char*`) of the target using the `get_direct_mem_ptr(...)` method. This is typically applied for memory components. In the case of the "fetch" mentioned above, once the direct pointer has been obtained, a fetch operation can be implemented with a simple access to a C++ array. Target modules other than memories are a bit harder to deal with, but one can obtain a pointer to the target module and call arbitrary public methods using ignorable payload extensions, or abusing DMI with unsafe type casts to transport a pointer to module.

DMI can be seen as a manual, partial inlining. In the example of Figure 3.9, applying DMI to the program would essentially mean replacing the `socket.write(addr, data);` statement in the CPU by a `mem[addr] = data` statement, copied from the RAM module. DMI is a manual, and potentially dangerous optimization. Any safety check performed in the interconnect (e.g., check that the address is actually mapped to a target module, check for address alignment, ...)

is bypassed. Indeed, the initiator only obtains a pointer to the storage, and it is its responsibility to ensure the validity of memory accesses.

The main goal of Tweto is to allow optimizations similar to DMI in effect, but in an automatic and therefore safe way. This is done by considering the architecture of the platform as static, which allows resolving virtual function calls, and inlining method calls from a module to another.

The starting point for Tweto was a draft written by Claude Helmstetter (Liama laboratory). It was written for SystemC base constructs, without support for TLM. Technically, it takes advantage of the LLVM JIT compilation: the elaboration phase of the SystemC program is executed, and an optimization pass is executed before the actual simulation starts. This optimization pass duplicates and specializes the body of each process. The duplication of process, together with some knowledge of SystemC, allows making some value constant (e.g., the value of `this` within a process is known and constant for each process, and the pointer to the interface contained within a port is also constant after the elaboration is done). These constant values allow further common optimizations like virtual method resolution and function specialization (e.g., turn `f(42, x)` into `f_42(x)`, and optimize `f_42`).

3.3.3 Integrating Tweto with PinaVM

When we started working on Tweto, with Si-Mohamed Lamraoui, the tool was already able to inline some function calls through ports. Our goal was to allow optimizing TLM communications through bus models, i.e., optimize cases presented in the motivation section above (3.3.1). A generic optimization that would work for any bus model would be hard, hence we chose one bus model (a simple example called *BASIC*, originally developed for Ensimag students), and wrote the corresponding optimizer.

PinaVM provides the technical basis for a TLM-aware version of Tweto: it allows running the optimizer after the elaboration phase, exposes the platform architecture, the address map, and the LLVM bitcode to the back-end. Tweto was integrated as a PinaVM back-end, and in the case of accesses to constant addresses, the address resolution was done in the optimizer. The transaction started in the initiator is replaced with a direct function call to the target module. The address processing done by the bus model (like converting absolute address to relative ones) is also done within the optimizer. Experiments on a very simple program showed promising results, with a speed-up by a factor of 5 in some cases. The tool needs more work to be applicable on real-life programs, but the preliminary experiments show that the direction is worth investigating.

To be effective in non-trivial cases, the optimization performed by Tweto should be combined with a value analysis. This would allow dealing with code like

```
addr = SOME_CONSTANT;
socket.write(addr, data);
```

where the address is actually a constant, but not passed as a literal constant in the code, or

```
for (int i = 0; i <= 10; i++)
        socket.write(SOME_OTHER_CONSTANT + 4 * i, data);
```

where the address is not constant, but can easily be bounded (which would still allow address decoding statically).

Part II

Non-functional Properties

Chapter 4

Non-functional Properties in Transaction-Level Modeling

> *"And then came the grandest idea of all! We actually made a map of the country, on the scale of a mile to the mile!"*
> *"Have you used it much?" I enquired.*
> *"It has never been spread out, yet," said Mein Herr: "the farmers objected: they said it would cover the whole country, and shut out the sunlight! So we now use the country itself, as its own map, and I assure you it does nearly as well."*
>
> — Lewis Carroll, *Sylvie and Bruno Concluded (1993)*

Simulation and transaction-level modeling is not only used to check the functional correctness of systems, but also to perform some preliminary estimations on non-functional properties, like time, power consumption and temperature.

The first section of this chapter deals with time, and proposes new tools to change the way time and concurrency is modeled, and to exploit this for better simulation performances. Then, Section 4.2 presents our work on modeling of power consumption and its effect on temperature, at several levels of precision and abstraction.

The distinction between functional and non-functional (sometimes referred to as *extra-functional*) properties is not always as clearcut as it may appear. Most systems have a closed-loop interaction between their functional and non-functional aspects: computations take time, consume energy and influence the temperature, and on the other hand the non-functional values influence the computation. The power-management strategy of a system performs a (functional) computation to manage the non-functional aspects of the chip. This document presents them it two separate parts, but the functional aspects should not be forgotten when dealing with the non-functional ones.

4.1 Time and Concurrency

4.1.1 Time and Concurrency in SystemC

Before presenting the contributions, we start with a brief reminder of the way time and concurrency are modeled in SystemC/TLM.

SystemC, like most discrete-event simulators, has a notion of *simulated time*, which corresponds to the time the actual system would take to perform the action being simulated (it is sometimes referred to as "SystemC time"). It is different from *wall-clock time*, which is the time the simulation takes to run. The relationship between simulated time and wall-clock time is

illustrated in Figure 4.1: computations obviously consume wall-clock time, but the simulated time only elapses when the program lets it do so (using `wait` statements). Time elapse occurs when processes and events programmed at the current instant have completed, and takes almost no wall-clock time (it is a simple variable increment). Concurrency is modeled using interleaving semantics. Two actions are concurrent if they are executed at the same simulated time, but SystemC will never run more than one process in parallel.

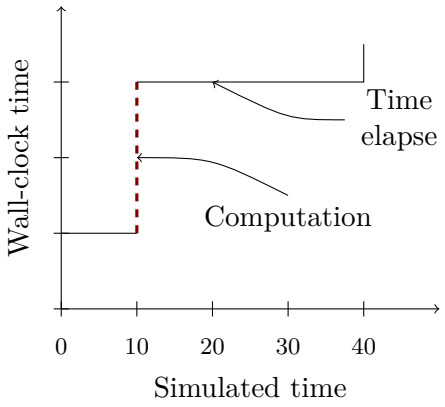


Figure 4.1: Simulated-time Vs wall-clock time

One relevant question is the choice of the points where the SystemC program should yield the control back to the scheduler to let other processes execute and time elapse. For *cycle-accurate* models, the choice is dictated by hardware clocks, but in the case of TLM, it is natural to consider large chunks of code as atomic, and avoid yielding too often. From the modeling point of view, the fine-grained implementation details in SystemC are often not relevant from the actual system point of view, hence testing fine-grained interleaving would not make much sense. For example, an image processing model may call a sequential function `process_image()` while the actual implementation would be highly parallel and pipelined. The individual lines of code of `process_image()` would have no correspondence in the actual system. Also, from the performance point of view, `wait` statements are costly hence should be avoided for faster simulation. On the other hand, a program that does not yield often enough creates the illusion of large atomic sections, and may not reflect all the behaviors of the actual system. Also, abstracting away internal details obviously reduces the timing precision: models using coarse timing granularity are often referred to as *loosely timed*.

This question is discussed in detail in [Cor08], which defines a notion of system synchronization points, which are points where the result of a computation become visible to the rest of the platform. The proposal of [Cor08] is to yield before each synchronization point. In other words, a process is decomposed into a set of steps, each being written along the lines of:

```
compute();
wait(duration);
commit();
```

The situation is depicted in Figure 4.2. `compute()` can be either a computation that is actually done in a private memory in the real system, or a computation for which intermediate steps in the model are not relevant with respect to the actual system. `commit()` is the operation that makes the result visible (e.g., sending an IRQ, writing to the control register of a target component or writing to a memory location polled by another process). Performing the `wait()` before the synchronization ensures that the `commit()` operation is executed at the right simulated time.

If the system receives an interrupt or a transaction during the computation, then in the model, it will be received while the `wait()` statement is executed, hence taken into account only

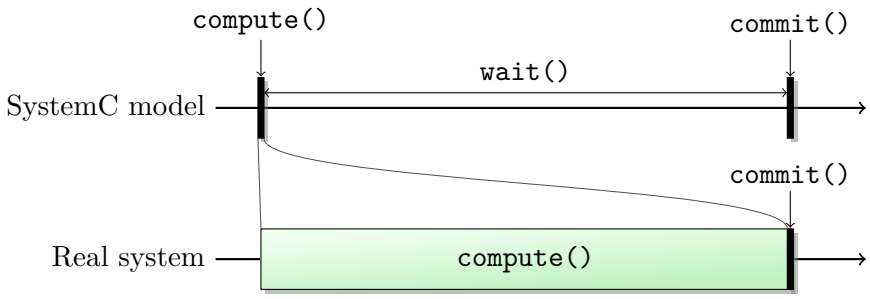


Figure 4.2: `wait()` Statements and Synchronization Points

after the computation has finished. This is not a limitation, but a modeling principle (sometimes referred to as "Cornet's principle"): either the programmer decided not to model the details of the computation, and the effect of communication is purposely delayed, or the programmer should have placed more synchronization points. For example, if the system receives an interrupt that cancels the computation, then a model that does the complete computation but cancels it right before the `commit()` is still faithful, even though it computed a bit more than necessary, since the functional effect is canceled.

The use of this principle has an important impact on the faithfulness of the model. It makes the comparison of high-level TLM programs and lower-level models (cycle-accurate TLM or RTL) a non-trivial problem (studied e.g., in Giovanni Funchal's Master II internship).

This principle, however, reaches its limits when the model includes concepts other than functional behavior and parallelism with interleaving semantics:

- One issue is when interleaving semantics is too strong compared to reality. For example, on most models, read and write operations to a RAM are considered atomic. Therefore, a process that sees the effect of a `commit()` also sees all the effects of the corresponding `compute()`. Real systems, however, may reorder some operations, and the programmer has to use extra synchronization primitives to ensure that this is the case. In case the software is badly synchronized, the model may not exhibit the faulty behaviors where the `commit()` is performed and the effect of `compute()` is not yet visible (e.g. computed in a cache which hasn't been flushed to the main memory yet). This would be a real bug in the actual system, but it would remain unnoticed on the model. This is the topic of Section 4.1.3.

- When modeling non-functional properties (power consumption or precise timing), the atomicity of long code section prevents modeling abortion of a computation for example: if interrupted in the middle, a task does not consume as much as if it was executed completely, and the reduced consumption should be reflected in the model. This will be dealt with in section 4.2.4.

4.1.2 Temporal Decoupling and Performance Optimizations

Temporal Decoupling in TLM-2

One particular case of Cornet's principle is *temporal decoupling*, defined in TLM-2 [Ope11] (although the second was not historically created following the first). In a temporally decoupled model, each process keeps a local clock t_{local}. Wait statements are replaced by increments over this t_{local}, and the local clock is synchronized with the global SystemC time from time to time. The scheme above is therefore replaced with:

t_{local} `+= compute_1();`

```
t_local += compute_2();
wait(t_local); t_local = 0;
// Possibly a commit() operation
```

An obvious advantage of this approach is that it reduces the number of `wait` statements in the model, hence reduces the slowdown due to context switches (which could dominate the execution time for fine grained models).

TLM-2 suggests synchronizing local time with SystemC time whenever the local clock becomes greater than a given value called the *quantum*. Another option is to synchronize before every synchronization points as explained in the section above. Of course, it is also possible to do both. Manipulating the local clock manually can be tedious and error prone, so one can define a temporal decoupling API along the lines of Figure 4.3.

```
void inc(sc_time d) {
    t_local = t_local + d;
}
```

```
void sync() {
    wait(t_local);
    t_local = 0;
}
```

Figure 4.3: Simple Temporal Decoupling API

One potential issue with temporal decoupling is that it can change the actual ordering of events. Consider the example in Figure 4.4 (to simplify the examples, we omit time units). The second process will execute `y = x` at time 11. The first process clearly has executed `x = 10` before, as it was done at time 10. In terms of local time, the statement `x = 15` is executed at time 15, but from the SystemC point of view, the current time is still 10, hence this `x = 15` is executed before `y = x`. The result is therefore `y == 15`. The timing diagram in Figure 4.4 shows actual SystemC instants in bold, and local times with thin vertical lines.

In this example, the reordering could be avoided by using explicit synchronization points (because `x` is a shared variable, assignments to `x` should be considered as synchronization points, and a `sync()` statement should be inserted before each of them). An alternative is to use a time quantum, which must be small enough (in our case, it must be smaller than 5). If the time quantum is used to ensure the correct synchronization of the platform, then choosing the right value for the quantum can be tricky: a small quantum can lead to slow simulation, and a large quantum will lead to bad synchronization and can prevent the simulation from running properly (see e.g., Figure 5 in [DGH+08] for an example "simulation performance Vs number of errors" trade-off). The right quantum can vary from program to program, but also from subsystem to subsystem or even from one simulated instant to another.

Note that while relying on a quantum for correct synchronization can be dangerous, a quantum-based decoupling can also be used to provide a reasonable timing accuracy. Indeed, the performance penalty in the actual system due to concurrent access to shared resources like the bus cannot be precisely modeled if the local clocks cannot be bounded. In this case, the trade-off becomes "simulation performance Vs timing accuracy", without affecting the functional correctness. This kind of trade-off is very common and well accepted in approximately timed models.

Smart FIFO

In general, temporal decoupling should therefore be used with care, and the mechanisms to ensure the correctness of the model, while keeping good performance are non-trivial to set up. An interesting case, however, is the one of temporal decoupling of processes communicating with FIFOs. I contributed to the *Smart FIFO* [HCG+13] initiated by Claude Helmstetter during his post-doc at CEA-LETI. It allows the use of temporal decoupling without having to rely on a

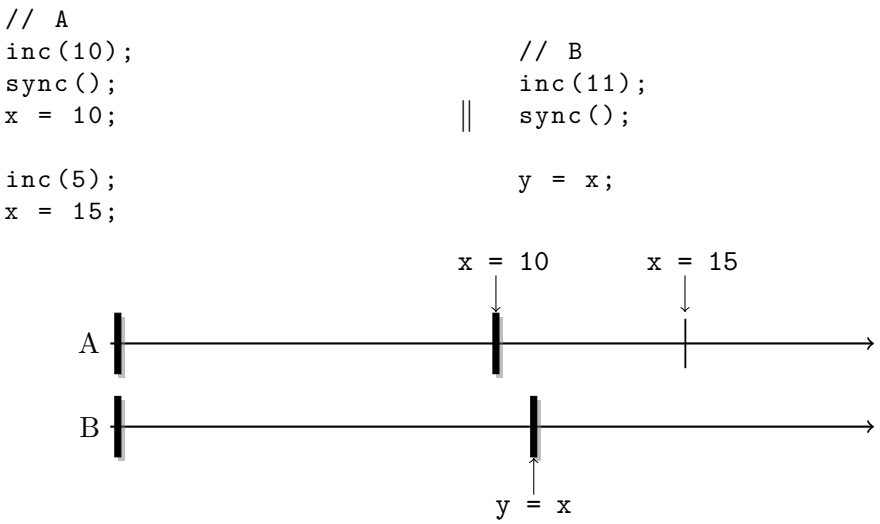


Figure 4.4: Example Reordering due to Temporal Decoupling

quantum or allowing reordering of events. The temporally decoupled models behave the same way as the non-temporally-decoupled ones. In simple models, this is relatively easy: the system behaves as a Kahn Process Network (KPN) and the timing does not influence functionality, so attaching timestamps to messages sent through FIFOs is sufficient: `write` statements may post messages in the future (i.e., with a timestamp greater than the current time), and `read` statements receiving a message with such a timestamp increase their local date to the timestamp received. Smart FIFO goes beyond the KPN model in several ways:

- It models bounded FIFOs, where `write` statements can be blocking. We consider empty cells in the FIFO as a resource with an associated timestamp. `read` operations consume one message and produce one empty cell with a timestamp, while `write` operations consume an empty cell and produce one timestamped message.
- It provides a monitoring API, including a `getSize()` function that returns the number of messages in the FIFO at a given time. The implementation of this function is non-trivial, as a message present in the FIFO may actually belong to the future. One timestamp per cell is not sufficient, since a cell may be currently occupied, but temporally decoupled processes may have been freed and filled again in the future, hence, checking the message creation timestamp is not sufficient. The smart FIFO therefore keeps both a "last insertion date" and a "last free date" for each FIFO cell.

It also provides a non-blocking interface so that `SC_METHOD` processes that cannot call SystemC's `wait` can use the FIFO.

4.1.3 Limitations of the Concurrency Model of SystemC/TLM

Modern processors use a lot of optimizations like cache, pipeline, prefetch, write buffers, which make the actual, internal execution of software relatively different from the intuition. For sequential programs, these optimizations are usually transparent: the processor (and compiler) may reorder instructions, but the end result will be *as if* the execution was sequential. Reading a variable may be done from a cache instead of the actual variable location, but the cache is meant to be consistent. In parallel systems, however, the same optimizations may yield surprising results. A common example is given in Figure 4.5. A naive reader may think that the

assertion holds, but it is usually not the case, as both the compiler and the hardware may notice the absence of dependency between `x = 42` and `y = 1` and rewrite the first process as `y = 1; x = 42`. Marking `x` and `y` as `volatile` in C or C++ would prevent compiler optimizations, but not hardware ones, hence is *not* a solution [Cor].

```
x = 42;                    ||    while (y == 0)
                                     continue;
y = 1;                             assert(x == 42);
```

Figure 4.5: Buggy program under a weak memory models (`x` and `y` are shared variable, initially `y == 0`)

Giving semantics to read and write operations to shared memory is therefore not a trivial issue. The set of allowed behaviors for a given execution trace of each thread is called a *memory model*. See for example [Boe05] for a detailed explanation of the problems. The ideal memory model, where each actual execution can be obtained by a system where each access is atomic and accesses are totally ordered is called *sequential consistency*. Each multiprocessor architecture comes with its own memory model, usually *relaxed* compared to sequential consistency. To allow programmers to write portable code, and to specify which compiler optimization is allowed, most modern programming languages (C++11, Java, ...) also define a memory model. For example, a correctly synchronized version of the program in Figure 4.5 in C++11 is given in Figure 4.6. Using the atomic `store` operation generates a *compiler barrier* that prevents reordering by the compiler, and forces the generation of a processor *atomic operation* if the processor's memory model requires it.

```
x = 42;                                  while (y.load(memory_order_acquire)
                                                == 0)
y.store(1,                         ||             continue;
    memory_order_release);
                                         assert(x == 42);
```

Figure 4.6: Correctly synchronized version of Figure 4.5 in C++11 (`y` is declared as `atomic<int> y(0)`)

SystemC gets rid of memory model issues by using sequential, *co-routine semantics*. This is convenient for SystemC programmers, since it avoids a number of bug opportunities. As memory model issues are usually very tricky to deal with, this is arguably a good thing. In particular, an important property of SystemC simulations is that they are reproducible.

Unfortunately, using a less error-prone programming model is not always a good thing when it comes to modeling. The model must not only be "correct" in the sense that it shows the desired behavior, it must also be *faithful* in the sense that it exhibits *all* the behaviors of the actual system. To illustrate the problem, consider a SystemC/TLM model used for embedded software development. If the SystemC/TLM removes bug opportunities, for example claiming that the hardware architecture is sequentially consistent, then this model will be "comfortable" for the software programmer, who may write incorrectly synchronized code, and still have working simulations. Unfortunately, this illusion of comfort would actually be a serious issue with the model, as the code written and tested with it would not work on the actual system.

The faithfulness issue is illustrated by Figure 4.7. The situation where the model exactly reflects the behavior of the actual system (i.e., (A) = (B) = $\emptyset$) is both hardly achievable and not desirable. The set (A) of extra behaviors of the model should be kept small, and should not

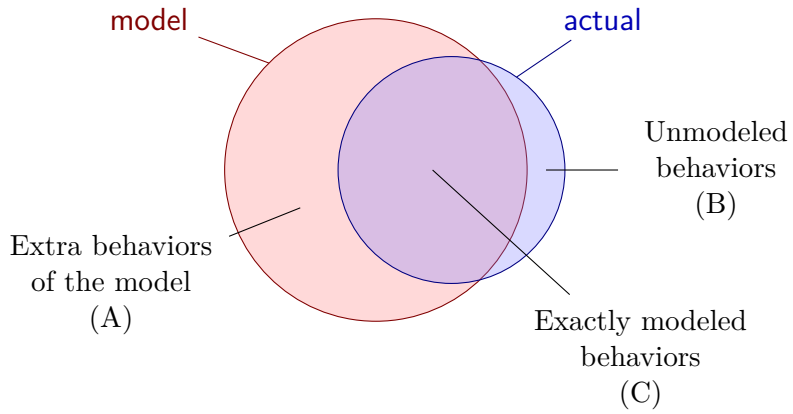


Figure 4.7: The faithfulness issue

contain unreasonable behaviors. It needs not be empty though: letting the system exhibit more behaviors than the real system is a way to ensure robustness. For example, a model with loose timing can exhibit embedded software bugs that would not occur on the real system because the particular timing of the actual system forces an ordering where the bug isn't triggered. The set of unmodeled behaviors (B) should ideally be empty, but some trade-off between faithfulness and abstraction level must be considered. As already discussed in section 4.1.1, TLM usually considers the model at a coarse granularity hence abstracting away some behaviors. Introducing non-determinism and over-approximations in the model is a way to keep the set (B) small with a high level of abstraction. For example, increasing the level of abstraction removes some details linked to timing, but using loose timing should ensure that all ordering are still possible in the model.

The *faithfulness* issue was the main research topic of Giovanni Funchal's Ph.D [Fun11]. The original subject was to define a notion of component at the TLM level. One contribution of the work was to show that the definition of a component has to include a definition of the memory model: the interface of TLM components relies on `read`s and `write`s on a bus, and defining their semantics requires a memory model. It is not sufficient to define the set of allowed `read`/`write` sequences: it is also necessary to define what will be actually visible (depending on the memory model, a `read` operation by component following a `write` by another component at the same address may or may not see the effect of the `write`). The identification of the problem with several supporting examples is presented in [FMMCM11]. The first conclusion is that the current modeling practice in TLM do not allow seeing memory model related bugs in embedded software.

We explored two directions towards a solution. The first is described in [Fun11], and consists in adding non-determinism in the interconnect model to reflect the effect of the memory model without modeling the details of the implementation (i.e., the actual cache, pipeline, write buffers, ...). The experiments were made on a *Total Store Order (TSO)* memory model, using non-deterministic FIFO in the style of [BP09]. The second direction was to change the concurrency model used in the simulator, by replacing SystemC with an alternate simulator described in the following section.

4.1.4 jTLM: an Experimentation Platform for Transaction-Level Modeling

Transaction-level modeling is usually done in SystemC, which is clearly the standard of the domain. As a result, it may be difficult to distinguish the "TLM abstraction level" and the implementation of TLM in SystemC. As we saw in the previous section, the concurrency model

of SystemC has a few particularities, and it is tempting to generalize the current constraints and practices in SystemC to transaction level modeling in general. After years of research with SystemC, we decided to write an alternate simulator called *jTLM*, to experiment with TLM concepts outside SystemC.

The guiding principle writing jTLM was to be as different from SystemC as possible. To avoid the temptation of reusing SystemC code, we chose a different host programming language (Java), and we started with a different concurrency model from the beginning. jTLM was started by Nabila Abdessaied, and then completely re-written by Giovanni Funchal and I.

Experimenting Preemptive Simulation in TLM

As opposed to SystemC, the execution parallelism in jTLM is built in the semantics. SystemC distinguishes runnable processes and a single running process (elected by the SystemC scheduler among the runnable processes). jTLM considers that a process is either stopped or "running". Running processes are scheduled by the Java virtual machine (JVM), they may actually run in parallel if the host machine has enough cores or be scheduled preemptively. Actually, the jTLM scheduler is a very thin layer on top of the JVM based on common discrete-event simulation principles.

As a consequence, the correct synchronization is left to the responsibility of the programmer. Unlike SystemC, jTLM forces the user to use Java's synchronization primitives (e.g., `synchronized` keyword) to access shared variables. Direct calls to Java's `wait` and `notify` are strongly discouraged as they would block a process without notifying the jTLM scheduler (hence prevent time from elapsing). New jTLM-specific primitives are provided to replace them, like the `Event` class. jTLM threads run in parallel on top of the Java Memory Model, which is arguably a good thing as it means that processes that omits important synchronization may exhibit faulty behaviors. It does not exhibit faulty behaviors due to memory models weaker than the Java memory model, though.

jTLM also comes with a cooperative mode (prototyped by Raphael Velasquez), where only one runnable process can be executed at a time, without preemption. The execution mode does not change the jTLM API, hence the same jTLM program can be executed in both modes. A correct program should run equivalently in both modes: accesses to shared resources should be protected, and progress should be ensured without relying on preemption (i.e., like in SystemC, potentially infinite loops should be broken with a statement that yields the control to the scheduler and lets time elapse). The preemptive/parallel mode of jTLM gives better performance since the simulation actually runs in parallel, and the cooperative mode provides reproducibility and forces the programmer to use the right primitives to ensure progress. The model of concurrency of jTLM is discussed in further details in [FM11a].

Modeling Duration of Tasks

Because the semantics of jTLM *is* parallel, nothing has to be done to parallelize the simulation when several processes are runnable in parallel. Still, this does not completely solve the parallelization issue: when only one, or a few processes are runnable at the same time, jTLM won't be able to exploit the host machine's parallelism. In addition to allowing processes to execute in parallel, an efficient parallelization scheme must provide ways to maximize the number of processes runnable at the same time.

This problem had already been identified long ago [Bou07], and is particularly important for high-level models. RTL or cycle-accurate models use hardware clocks. One tick of the hardware clock normally wakes up several processes, which execute at the same time, and are good candidates for parallelization. It is not the case in TLM, which often uses quantitative or even loose timing. The odds of two processes running `wait(`*`some-constant`*`)` and

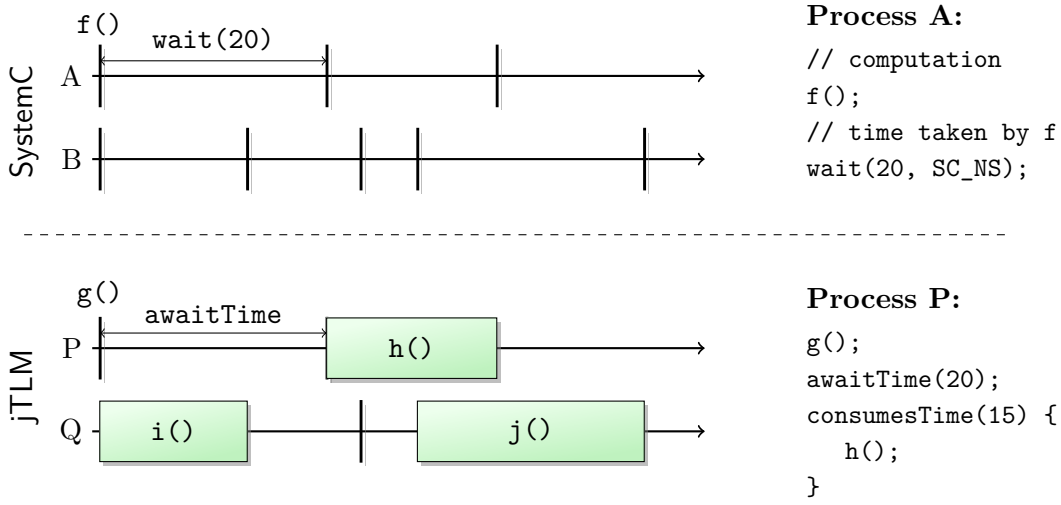


Figure 4.8: Time and Duration in SystemC and jTLM

`wait(`*`some-other-constant`*`)` waking up at the same SystemC instant are small. The ones of two processes running `wait(random(...))` doing so is even smaller.

jTLM's proposal to solve this problem is to allow *tasks with duration* [FM11b]. As discussed in Section 4.1.1, the notion of duration is usually present in the mind of programmers, but is expressed in SystemC as "`compute(); wait(`*`duration`*`); commit();`" sequences. The programmer knows that the `wait()` statement corresponds to the actual duration of `compute()`, but the SystemC kernel can't exploit this. In jTLM, a computation that takes time is expressed using the `consumesTime` primitive[1]:

```
consumesTime (duration) {
    compute();
}
commit();
```

The execution of a jTLM program including `consumesTime` constructs is illustrated in Figure 4.8. The SystemC way to express time is still available with the `awaitTime` construct.

jTLM guarantees that the code of `compute()` is executed between the time t where the `consumesTime` call is started, and $t + duration$, but not more. `compute()` may be executed in parallel with any other code executed in the interval $[t, t + duration]$. This means that *simultaneity* is no longer necessary for parallelism, as task *overlap* in time is sufficient.

One limitation of `consumesTime` is that the duration of the task must be known before starting the task (unlike the SystemC equivalent which executes the `wait` statement after the computation, hence can use data collected during `compute()` in the argument passed to `wait()`). With Mohamed-Taoufiq El-Aissaoui, we proposed an alternate primitive called `consumesUnknownTime` which does not require the duration to be known. Instead, the end date of the task is the date when the execution of the task completes.

In preemptive/parallel mode, `consumesTime` and `consumesUnknownTime` are implemented within the jTLM scheduler. Although they have very different semantics, their implementation is surprisingly similar to the ones of `awaitTime`: in both cases, the implementation uses the "time queue", a priority queue containing all the events programmed in the future. The primitive `awaitTime(`*`duration`*`)` programs the process wake up at time $now + duration$ and acquires a semaphore to blocks the current process until the wake up event is triggered. The

[1]This is pseudo-code to simplify the example. The actual code uses anonymous class and requires a few more pairs of braces.

primitive `consumesTime(`*`duration`*`)` does the same, but passes the control to the user code before acquiring the semaphore. `consumesUnknownTime` simply removes the current process from the list of processes executing at the current time before executing the user code, and re-adds in afterwards. In other words, the process does not prevent time from elapsing while executing `consumesUnknownTime`.

`consumesTime` also has an implementation in cooperative mode, where it basically executes the user code instantaneously, and an `awaitTime` statement to model the duration (it can actually split the interval into two pieces, and execute the user code instantaneously at any date within the interval). `consumesUnknownTime` has no implementation in cooperative mode for now.

jTLM was a nice research experiment. A very good property of the current implementation is that the code is very small (less than 600 lines of code including the scheduler, a bus and a RAM model), which makes it very easy to experiment new ideas. However, it was never meant to be used in production or compete with SystemC. The next logical step is therefore to find out which of jTLM's idea can be adapted and benefit to SystemC.

4.1.5 Parallelization with `sc-during`: Adapting jTLM's Ideas to SystemC

The main motivation for transaction-level modeling was simulation speed. Abstracting away implementation details led to a tremendous speedup from RTL or cycle-accurate simulations. However, the complexity of hardware designs keeps increasing following Moore's law, while the speed of individual cores of a typical workstation has not evolved much during the last decade. To catch up with the increasing complexity of the system being modeled, one needs to exploit the host parallelism, which is not possible with SystemC's reference simulator, and actually made hard by SystemC's reference semantics which enforces co-routine semantics.

Attempts have been made to parallelize SystemC simulations without changing the semantics. [Bou07] proposed a static analysis for SystemC together with a scheduling algorithm that ensures that two processes running at the same time will not touch the same variables. [CHD12] applies the same ideas on SpecC. These approaches can work well on low-level models (typically cycle accurate ones). First, processes yield very often, hence access very few variables during each transition, hence the number of conflicts between threads should be low. Second, many threads will usually be runnable during each cycle, so the parallelism opportunity should be high.

Applying the same techniques for TLM programs on the other hand is very inefficient. Because communications in TLM are done using function calls from a module to another, variables in target components accessed during a transaction are considered as accessed by the initiator process. In particular, a RAM component can be seen as one global variable shared by all processes accessing it. Worse, when a transaction is performed on a bus, if the address cannot be resolved statically, then all the target modules connected to the bus are considered as shared variables by all initiator processes. A static analysis will therefore have to consider that most processes are in conflicts with each other, or perform advanced static analysis (the analysis performed by Tweto in Section 3.3 would be a first step).

Additionally, one of the lessons of the jTLM experience and of [Bou07] was that relying on instantaneity for parallelization was not sufficient. [CD13] extended the scheme proposed in [CHD12] to allow parallelism across cycles, but works only for cycle accurate simulation. Applying semantics-preserving techniques to transaction-level models does not seem realistic.

An alternative is to change the semantics of SystemC, and require extra-synchronization in the user code as it is the case in jTLM. There are a number of implementations of parallel SystemC kernels, like [SLPH10] following this approach. Unfortunately, changing SystemC's semantics means that legacy code may stop working properly, which would not be acceptable in

an industrial context.

We experimented another direction: instead of trying to parallelize the user code without modifying it, we provide the programmer new primitives to express concurrency. Legacy code can continue working unmodified, but without performance benefits. Performance-critical code can be modified to run in parallel with the rest of the simulation, but may require extra synchronization code. In summary, we need to provide two categories of primitives:

desynchronization primitives, to relax the total order imposed by SystemC, and allow one process to run ahead of or behind the current time of other processes. The `consumesTime` primitive of jTLM is an example.

synchronization primitives, to ensure that the desynchronization introduced above does not prevent the simulation from running properly. In jTLM, we relied on Java and the keyword `synchronized`, and provided classes like `Event`.

Adding Synchronization and Desynchronization within SystemC

A first attempt to provide *desynchronization* primitives was carried out with Samuel Jones. The tool developed during his internship is a modified version of SystemC that allows partitioning the system being simulated. Each *partition* runs its own scheduler, in their own operating system thread. This way, communication within a partition can be done the usual SystemC way, and only inter-partition communication needs extra care. A system made of only one partition behaves exactly like a standard SystemC program.

Each partition also has its local time, hence, a partition is also called a *time warp*. The local times of time warps can evolve independently, but can also be synchronized by the programmer. One way to synchronize local times is to define a *global quantum*, which will act as a bound over the differences between the local times of any two time warps. The tool also allows explicit synchronization between two warps (a warp $C_{initiator}$ can request a synchronization with another warp C_{target}). There are several kinds of synchronization, including:

`SYNC_CATCH_UP` blocks C_{target} and lets the simulation continue in $C_{initiator}$ until it reaches the same local time as C_{target}.

`SYNC_WAIT` does the opposite: it blocks $C_{initiator}$ until the local time of C_{target} reaches the same value.

`FULL_SYNC` does both `SYNC_CATCH_UP` and `SYNC_WAIT`.

Synchronization is needed only by communications, hence we integrated the synchronization primitives with the communications. We implemented this approach in the BASIC convenience API (very simple API already mentioned in Section 3.3.3). `read` and `write` functions are extended to take an optional argument describing the kind of synchronization (`SYNC_CATCH_UP`, `SYNC_WAIT`, ...) to perform before actually processing the transaction.

We experimented the approach with Henry-Joseph Audeoud by partitioning the "game of life" example presented in Section 1.2.1. The results were disappointing although not surprising: the program was running at about the same speed, but using two CPUs instead of one! This is explained by the fact that the implementation focused on semantics and choice of the right synchronization primitives, but not on performance, hence the core of the scheduler uses coarse-grained locking. Unsurprisingly, the approach also showed very good performance on embarrassingly parallel application with coarse granularity.

Partitioning a simulation required some modification of the SystemC scheduler. This work can be seen as preliminary experiment to propose new evolution directions for the SystemC official semantics. We believe that SystemC has to evolve to include parallelism, but also think that such evolution has to be done remaining backward compatible with today's version. We do not have a ready-made solution, but we hope that the idea of partitioning and the proposed

```
void during(sc_core::sc_time duration,
            std::function<void()> routine) {
①    std::thread t(routine); // create thread
②    sc_core::wait(duration); // let SystemC execute
③    t.join(); // wait for thread completion
}
```

Figure 4.9: Naive implementation of `during`

synchronization primitives will be a source of inspiration. In the short term, the partitioning idea can be useful to user having complete control over the tools they use, but is not acceptable in a context where a particular SystemC implementation must be used (e.g., Commercial implementations like Cadence, Synopsys or Mentor's versions, which are not open-source).

`sc-during`: Adapting the Duration idea from jTLM to SystemC

In contrast to the approach of Samuel Jones presented above, this section presents an approach that does not need any modification to the SystemC kernel. It is strongly inspired from jTLM's idea of tasks with duration.

jTLM's tasks are implemented directly in the scheduler, interacting with low-level data structures like the kernel's time queue. We wanted a non-intrusive approach in SystemC, to allow the user to benefit from the approach with any SystemC implementation. This was first partly implemented by Mohamed Zaim Wadghiri, and then re-written in the `sc-during` library [Moy12b, Moy13].

The library allows delegating some computation to an operating system thread (e.g., a pthread) that runs in parallel with the main thread executing the SystemC scheduler. The delegated computation corresponds to a task with duration in jTLM and is called a *during task*.

The idea behind `sc-during` is similar to Esterel's `exec` statement [BRS93, Ber00], which allows controlling an asynchronous task execution from a synchronous kernel. During tasks are executed outside the SystemC kernel, and by default execute without any synchronization with SystemC.

A naive implementation of `during` in C++11 is given in Figure 4.9, and an execution of a `during` call is provided in Figure 4.10. First, a new OS thread is created, executing the routine. Then, a call to SystemC's `wait` blocks the current SystemC thread while the routine is executed. This does not block the complete SystemC simulation, but lets SystemC's time elapse and other SystemC processes execute. Finally, a `join` operation ensures that the routine is completed and the simulated time has reached the right value when the `during` call completes.

This naive implementation has at least two drawbacks:

1. It creates a new thread for each task and destroys it afterwards, which can be costly in terms of performance, and
2. It does not allow any synchronization within the `during` task, so it only works when the routine has no interaction with SystemC's time and scheduling.

To solve the first problem, `sc-during` provides several strategies to reuse OS threads from a `during` task to another, either by using a pre-allocated set of threads or by instantiating one OS thread per SystemC thread that requires one. Instead of killing threads when the task is complete, they are blocked on a condition variable, waiting for another task to process.

To solve the second problem, a synchronization monitor is created for each OS thread. The overall structure is depicted in Figure 4.11. Each synchronization monitor (`sync_task`) contains shared variables protected with mutexes for communication, and condition variables for synchronization.

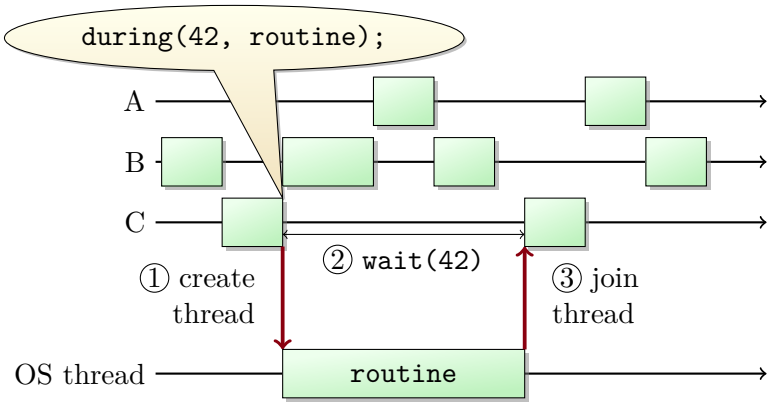


Figure 4.10: Execution of a `during` task (SystemC thread C calls `during(42, routine)`)

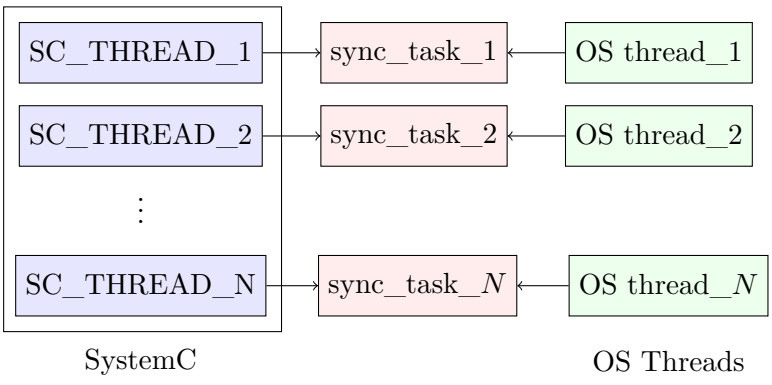


Figure 4.11: `sc-during` Implementation

sc-during and Synchronization

during tasks are a form of *desynchronization* primitives as they allow a routine to execute in its own thread, regardless of the SystemC time and scheduling policy. In complement, sc-during provides several *synchronization* primitives, to allow both synchronization of data and time. These primitives are meant to be called from a during task:

sc_call(f) allows the function f to be executed in the context of SystemC, thus allowing f to access variables shared with other SystemC processes and to use SystemC primitives like event notifications.

catch_up() blocks the current task until the SystemC time has reached the date when the task should end.

extra_time(t) increase the duration of the current task by t.

sc-during does not provide a direct equivalent of jTLM's consumesUnknownTime, but a during task may periodically call extra_time(), leading to an unbounded task (that will end only when the SystemC simulation terminates). We believe the sc-during solution is better since using catch_up() and extra_time(t) allows some control on the fairness between the during task and the SystemC simulation. consumesUnknownTime was completely relying on the fairness of the JVM's scheduler to ensure that both the consumesUnknownTime task and the rest of the simulation progress. The sc-during solution controls the simulated time at which various portions of the during task can be executed, akin to Samuel Jones' *global quantum*. For example, if a during task of initial duration d executes this piece of code

```
while(true) {
    f();
    catch_up();
    extra_time(d);
}
```

then the function f will be executed once in each interval $[k.\mathtt{d}, (k+1).\mathtt{d}]$ $(k \in \mathcal{N})$. On the other hand, the schedulers have the freedom to execute f at any time during each interval.

Controlling the time interval in which a piece of code is executed, but allowing a bounded jitter reminds temporal decoupling, introduced in Section 4.1.2. Indeed, the sc-during synchronization primitives map very well to temporal decoupling style. The temporal decoupling API defined in Figure 4.3 and recalled in Figure 4.12.(a) can be adapted when used inside a during task as Figure 4.12.(b). The semantics are not equivalent, but similar: in both cases, we rely on the fact that the exact execution date of a piece of code is irrelevant, but we control in which time interval the execution happens.

```
void inc(sc_time d) {
    t_local = t_local + d;
}

void sync() {
    wait(t_local);
    t_local = 0;
}
```

(a) traditional temporal decoupling

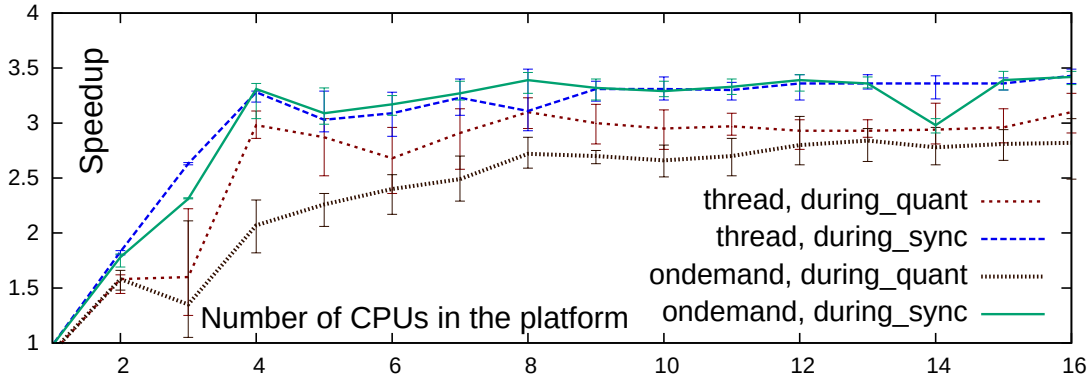
```
void inc(sc_time d) {
    extra_time(d);
}

void sync() {
    catch_up();
}
```

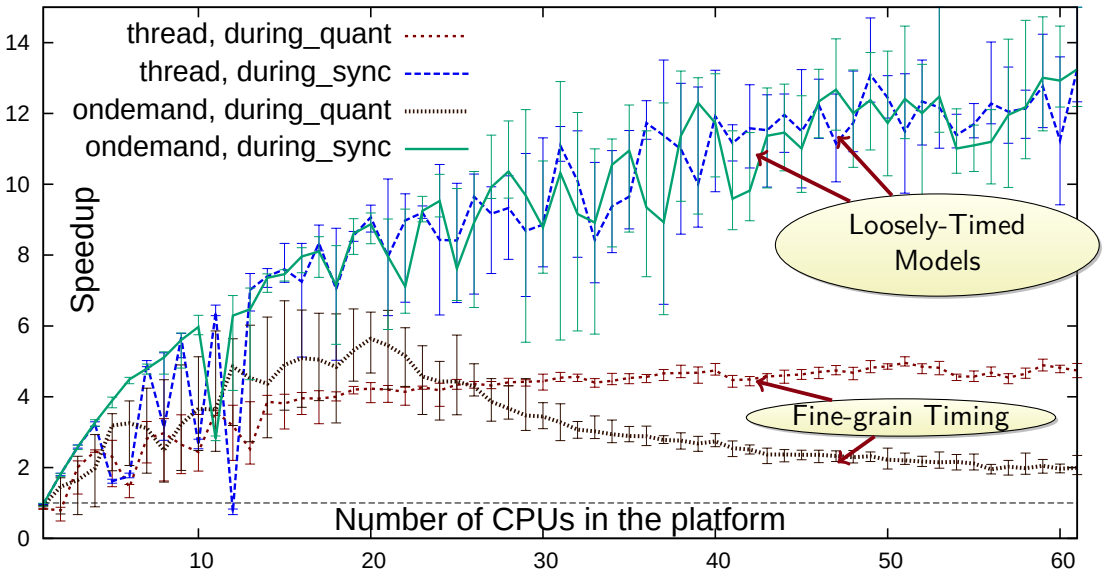
(b) temporal decoupling with sc-during

Figure 4.12: Simple Temporal Decoupling API

Borrowing ideas from Samuel Jones, we use the interconnect for synchronization. We modified the bus model to make it thread-safe (which is not difficult: the address map is read-only during execution, and other data can be local variables) and we made sure it does not use SystemC's `wait` and `notify`. This way, transactions that are known not to access shared variables can be executed without synchronization. On the other hand, some address ranges are marked as requiring synchronization, and bus accesses to these addresses are wrapped in a `sc_call`.



(a) Execution on a 4 cores machines



(b) Execution on a 48 cores machine

Figure 4.13: Speedup Compared to Sequential Simulation for Different Implementation (average, min and max over 10 runs)

Experimental results on the "Game of life" platform are shown in Figure 4.13 for different `sc-during` strategies and different uses of the library. A good surprise is that the naive implementation that creates a thread for each task (labeled "thread" in the figure) is not so bad and sometimes even performs better than the "ondemand" strategy that reuses the same thread for multiple tasks. So, the operating system is good at creating and deleting threads. For programs with limited parallelism (e.g., 4 or 8 `during` tasks in parallel), the acceleration is almost linear, so the results are satisfactory. The performance stagnate around an acceleration factor of 10, even for highly parallel programs. When the SystemC program uses fine-grained timing (hence calls `sc-during` primitives a lot), the speed even decreases after a certain point.

The `sc-during` library was written with software engineering's good practice in mind. The code is well tested (twice more code for automatic testing than for the implementation itself), and the codebase is kept small to ease maintainability (just above 1000 lines of code). It is open-source and available from `http://sc-during.forge.imag.fr/`.

Performance and Future Optimizations

These disappointing performance cases can be explained in several ways. First, we developed `sc-during` with the basic thread synchronization primitives (mutexes and condition variables essentially), and avoided busy-waiting. This has the nice property of not loading CPUs when not necessary, which means the approach will work nicely on machines shared between several users. The bad performance on fine-grained timing is not surprising since the approach explicitly targeted high-level models with loose timing, where other approaches do not apply.

Still, the situation could clearly be improved. We started profiling mutex contention in the library with Xavier Poczekajlo to identify the performance bottlenecks. The library has been ported to C++11's thread library by Francesco Bongiovanni and I (it was previously using `boost::thread`). Using C++11 will allow using atomic operations where applicable in a portable way (either to replace mutexes protecting individual variables, or to introduce busy-waiting where condition variables are too expensive).

4.2 Power and Temperature Estimation

Non-functional properties are not limited to timing. Other parameters such as power consumption and temperature are also of growing importance.

4.2.1 Power Management in Modern System-on-Chips

Power consumption of a silicon chip comes mainly from two sources:

dynamic power comes from switching activity. One can see transistors and wires as small capacitors that are filled-in and emptied when the corresponding value switches from 0 to 1 or 1 to 0. This consumption therefore depends on the capacity of physical elements (hence, of the physical process), and on the frequency at which values change, i.e., how much computation is being performed by the component.

static power comes from leakage current. For a given hardware component, it depends on the state of the power supply (on or off, and voltage), but is independent of the computation being performed. The leakage current tends to increase when the size of transistors is reduced. Since heating silicon decreases its resistivity, the static power also increases with temperature.

A simple way to reduce dynamic power consumption is *clock gating*, which stops the switching activity by stopping the clock. Clock gating can be applied automatically by the synthesizer in many cases, and the problem has been studied extensively for decades (e.g., [AMD+94, BDM96]). Clock-gating, however, is not sufficient in modern Systems-on-Chip, where static power is non-negligible, if not dominant.

To reduce static power, heavier techniques have to be used. *Dynamic Voltage and Frequency Scaling (DVFS)* allows reducing the voltage, hence both static and dynamic power, but requires a lower frequency. Switching from a voltage/frequency couple to another takes time, hence the policy behind DVFS is non-trivial, and is usually implemented in software (for example, the `cpufreq` drivers in Linux count for around 20,000 lines of code). Another technique is *power gating*, which consists in disabling the power supply of unused components. Power gating also

takes time, and the transitions from a mode to another may consume a lot of energy, hence the policy to chose whether to power-gate a component or not is again non-trivial. In the sequel, we call *power controller* the set of hardware components that directly control these physical mechanisms, and *power manager* the implementation of the policy, usually done at least partially in software. Bugs related to power management, or *power bugs*, can drastically increase the power consumption of a system, and discharge the battery of a mobile phone in a couple of hours [PHZ12]. Worse, serious bugs in the power management policy can lead to deadlocks, e.g., waiting for interrupts from a component which has been completely switched off.

The need for low-power is obvious for battery-powered devices, but it is also important for permanently plugged ones like set-top-boxes: in addition to being environment-friendly, a low-power system needs cheaper packaging and avoids the need for a noisy cooling system. Power consumption cannot be evaluated precisely independently from temperature, as hotter silicon consumes more (and consuming more in turn increases temperature).

Temperature peaks can physically damage the chip. Since high temperature increases silicon conductivity, hence static power consumption, a temperature peak can lead to a consumption peak, and therefore a voltage drop. If the voltage goes below the appropriate threshold, the system may stop working. Decoupling capacitors can be used to allow short consumption peaks, but not for longer ones. It is therefore critical to have power and thermal estimations before manufacturing the real system to dimension the packaging and cooling system appropriately.

Also, high temperature gradients lead to faster ageing of the chip [CM13]. To avoid peaks and minimize gradients, the chips are usually equipped with several temperature sensors (typically one or more per core on a many-core platform [MBF+12, BCTB13]), allowing the implementation of a software control loop to regulate temperature.

4.2.2 Virtual Prototyping for Power Aware Systems

Non-functional Models

A chip's non-functional aspects can be modeled by specifying the power consumption of individual components, the layout of components (called the *floorplan*), and the physical parameters influencing heat dissipation (from a component to another, and from the chip to its environment).

For system-level simulations, modeling individual transistors precisely is clearly too low-level and too slow. The physical parameters have to be abstracted into a set of operating modes for each component, often called *power-state* [BHS98, BJ03]. A power-state defines the power consumption (in Watts, or the current intensity in Amperes), possibly as a function of temperature. The energy consumption (in Joules, or the charge transported in Coulombs) is obtained by integrating the power over time. When the power consumption of a power-state is constant, the integration is simply a multiplication by a duration. If the set of power-state is finite, then the underlying formal model is the one of *linear-priced timed automata* [BFH+01].

A power-state is defined by several parameters:

Activity defines the computation being performed by the component (e.g., waiting, computing, ...).

Electrical state is the voltage and frequency of the component. It is influenced by DVFS and power-gating, and controlled by the power-controller.

Traffic is the amount of data processed by unit of time (e.g., number of transactions routed for a bus, number of reads and writes for a memory, ...). It has to be modeled separately, since the traffic of a component usually comes from the activity of another component.

Depending on the stage in the design-flow, the power consumption associated with a power-state can come from different sources. At early stages of development, they can be target values (i.e., that further development will consider as an objective), or extrapolation from previous generations of the same component. Later in the design flow, they can be more precise estimations based on RTL or gate-level simulations, and physical measurements when a prototype of the chip is available. Our contribution is not to provide these parameters to the user, but to allow exploiting these per-component parameters for a *system-level* simulation including power and thermal management policy and software. Because of heat dissipation, and of the relationship between power and temperature, the composition of the component's parameters is non-trivial.

Tools like *Docea Power*'s *ACEplorer* [KMS10] allow modeling a chip, taking into account both the power consumption and the thermal aspects. The traditional way to use ACEplorer for simulation is to define scenarios that define the sequence of modes for each component. The tool computes the power consumption and the evolution of temperature (including the feedback of temperature on power described above). Scenarios can be provided by hand, using UML's activity diagrams, or can be produced by a SystemC/TLM or RTL simulation.

In the scenario-based usage, the model does not allow the execution of power-aware software: the SystemC or RTL simulation provides non-functional stimuli, but has no access to the result of the power and thermal simulation. A temperature sensor, or battery monitor component could not be modeled. One of the goals of the HeLP ANR project was to allow such closed-loop simulation.

Software Execution on a Virtual Prototype

To validate these power and thermal managers early in the design flow, one needs virtual prototypes that allows execution of power-aware software. Among these models, various degrees of precision can be achieved:

Purely functional: Even a purely functional prototypes need models of temperature sensors: if the embedded software reads a value from one of its registers, then the simulated platform should include a component mapped at this address and returning a sensible value (possibly an arbitrary constant).

Non-functional contracts: Some basic, but yet serious mistakes like reading from or writing to a component which is switched off can be caught by assertions in the model. The assertions form the contract [MPA11] of the components, or of the hardware platform with respect to software.

Scenario-based models: To test some basic power management policies, one may need the non-functional values to take different values. For example, if a platform triggers an emergency stop when temperature goes above some threshold, then testing this functionality requires a scenario where the temperature returned by the sensor crosses this threshold. This can be done by returning a pre-defined sequence of values in the temperature sensor model.

Approximate models: When the policy to be tested is non-trivial, manually writing scenarios is not feasible anymore. One needs an automated way to get reasonable sequences of values. A simple thermal model can be sufficient: it will typically let the temperature decrease when the system saves energy and vice versa. For example, a software developer implementing a simple hysteresis policy (switch to power saving mode when temperature is too high, and switch back to normal mode when the temperature crosses a low threshold)

```
// SystemC thread
void compute() {
    while (true) {
        set_pwr_state("activity", "idle");
        wait(event);
        set_pwr_state("activity", "run");
        f();
        wait(100, SC_NS);
    }
}
```

Figure 4.14: Simple Power-model of a Hardware Component (pseudo-code)

State	Static Power	Dynamic Power
`idle`	$80.e^{0.2(T-300)}$ mW	0 mW
`run`	$80.e^{0.2(T-300)}$ mW	100 mW
`deep sleep`	0 mW	0 mW

Figure 4.15: Simple Power Model (numbers provided as a rule of thumb)

can use this model to test that the mode switch are correctly performed.[2] These models allow detecting some of the non-functional bugs in the embedded software (failure to enter a low-power mode, polling instead of explicit wait for an interrupt, ...).

Precise models: To validate the parameters of a power management policy and get the actual values for maximal temperature peaks and gradient, one needs a precise model. This implies precision in the timing of the platform, and on the thermal model of the chip. Some degree of precision is also needed to compare the efficiency of several power management policies.

We are interested in the last two. The following sections describe new tools to allow power and thermal modeling on SystemC/TLM at different levels of timing granularity. In both cases, the approach enriches a functional model with some power-state information, and co-simulates it with a dedicated non-functional solver.

4.2.3 Cosimulation of a Functional Simulator with a Non-Functional Solver

To illustrate the cosimulation concepts, let us first consider a very simple component, with 3 activity states. The component waits for an event in `idle` mode, and on reception of the event does a computation `f` that lasts 100 nanoseconds in mode `run`. A SystemC thread describing this behavior is provided in Figure 4.14. The power instrumentation corresponds to the two `set_pwr_state` calls. The instrumented model has to be accompanied with a power model, like the one in Figure 4.15.

The principle of cosimulation is illustrated in Figure 4.16. The instrumented functional model computes, at each point in time, the power state of each component. This power state information is transmitted (a) to the power model, which uses this power state information together with the temperature T obtained from the thermal solver (c) to compute the power consumption of each component. The power consumption is used by (b) the thermal solver solver to compute the derivative of the temperature. It is used together with the thermal model

[2] we actually had a bug where the low threshold was not properly dealt with in our case study. The bug remained unnoticed until a thermal model was added to the platform.

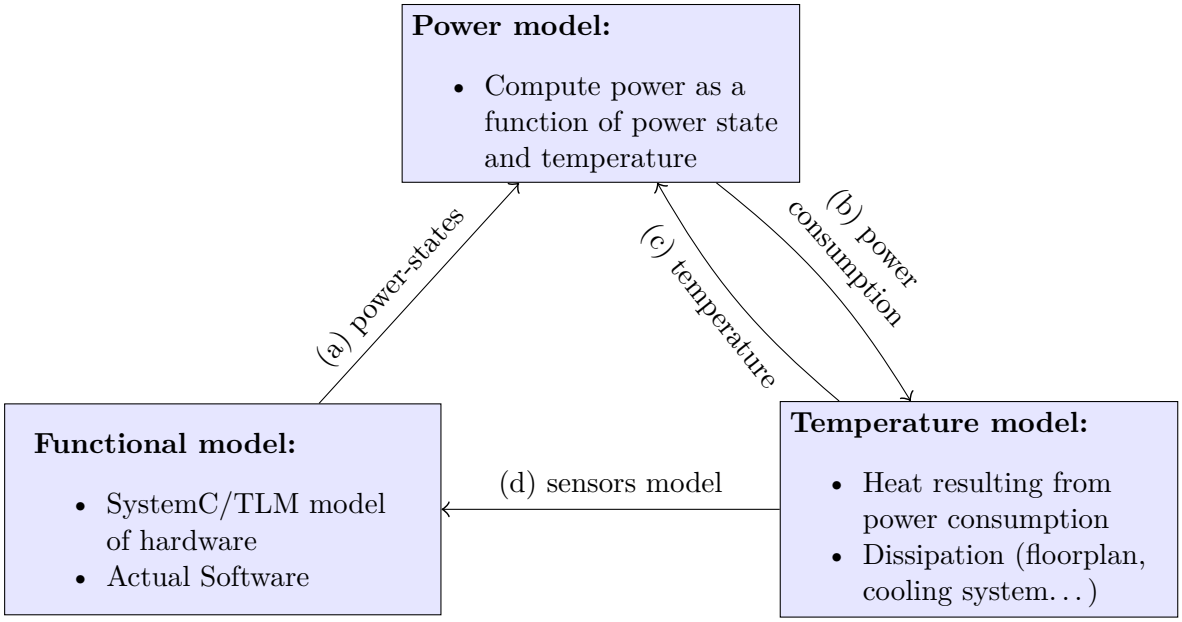


Figure 4.16: Functional, Power and Thermal Models

to compute the evolution of temperature. The temperature (and possibly other non-functional information) are transmitted back to the functional part and embedded software (c) through models of physical sensors.

One instance of these principles is the tool LIBTLMPWT, developed by Claude Helmstetter, with some technical help from Eduardo León. It is a library for use with SystemC, which uses *ATMI* [MS07] as the temperature solver. The power model is defined by the user the user provides the static and dynamic power consumption in each state as C++ functions that take the power parameters as arguments and return the power consumption values. The exchange (a, b, c, d) between functional and non-functional simulators follow the rhythm of ATMI. Since ATMI uses a fixed time-step, synchronizing SystemC and ATMI's simulated times is easy: the SystemC program includes a process executed with the same period as ATMI's time step. This process collects power-state information from all the components of the platform and turns these power-state information into power consumption by calling the power model's function. It then performs a step in ATMI (ATMI being a library, this is simply a function call). After this, the sensors' values are updated accordingly. LIBTLMPWT can display the results of simulation interactively with a GUI, as shown in Figure 4.17.

LIBTLMPWT allows using the basic TLM optimizations like DMI and temporal decoupling. The DMI management requires a minor instrumentation of the code to call functions of the power model for each DMI transaction. The management of temporal decoupling simply use the local time instead of the SystemC time in the power model.

LIBTLMPWT was written to be usable outside Verimag. It is open-source and available from `http://www-verimag.imag.fr/~moy/?LIBTLMPWT-Model-Power-Consumption`.

4.2.4 Functional and Non-functional Models and Loose Timing

LIBTLMPWT was written for model with a relatively precise timing, hence models whose SystemC time step is small. A typical approximately timed platform, even with temporal decoupling, will have a time-step around one microsecond. On the other hand, ATMI does not require such a small time step: experience shows that a time step of 1 millisecond is usually sufficient to achieve a good precision. In this context, it is acceptable to consider that an action

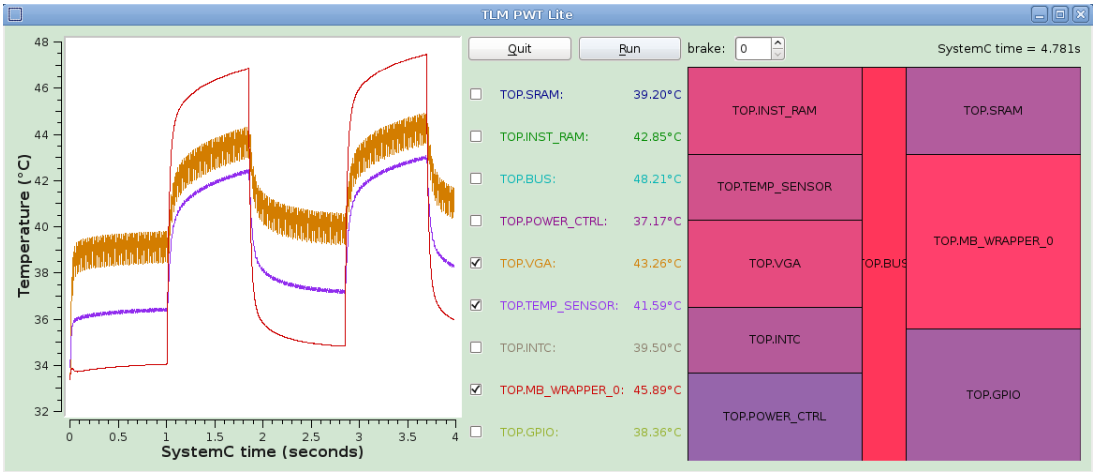


Figure 4.17: Graphical User Interface of LIBTLMPWT

modeled as `f(); wait(t);` (following the guidelines of Section 4.1.1) is actually an immediate computation `f` followed by an inactivity period of duration `t`, since `t` is small enough. The traffic generated by `f` on other components will therefore be taken into account by the thermal solver at the simulated time when `f` is executed. In case the thermal simulation triggers a *non-functional event* (e.g., the temperature crosses a threshold and an interrupt must be triggered), the non-functional event will occur at the next ATMI step.

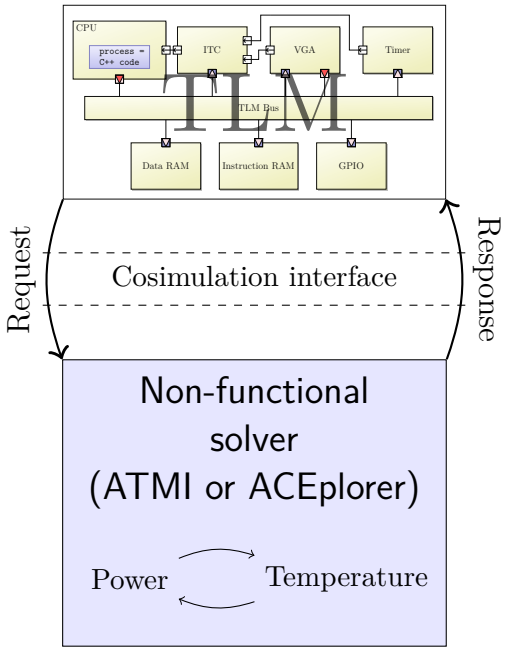
In contrast, this section provides solutions for models where the timing granularity of the functional model is coarse, and the time-step of the non-functional solver is not fixed. This research was driven by industry needs within the HeLP project: STMicroelectronics uses essentially loosely timed system, and the tool ACEplorer has a variable time-step, which is not exposed to the user. ACEplorer has to be used as a black-box, and a well-defined *cosimulation interface* has to be defined to let SystemC and ACEplorer communicate. We defined a Thrift-based [SAK07] interface, and several strategies using it. The interface is very generic, and is tied neither to SystemC nor to ACEplorer (indeed, we first experimented it with ATMI and HotSpot [HMG+06]). This work is presented in details in [BMM+13b].

Cosimulation Interface

When using the non-functional solver as a black-box it is no longer possible to execute a SystemC process for each of its internal steps. Instead, SystemC's time drives the simulation. From time to time, the SystemC program requests a non-functional simulation on a time interval. The non-functional solver normally performs the simulation and returns the relevant values at the end of the time interval. The cosimulation interface uses a simple request/response protocol illustrated in Figure 4.18. It relies on Thrift [SAK07] for inter-process communication.

The same interface can be used with multiple strategies. In the *lockstep strategy* (Figures 4.19 and 4.20), the functional/non-functional synchronization is performed at the end of each SystemC simulated instant. A non-functional simulation is requested for the time interval between the current instant and the expected next simulation instant. In case a non-functional event is triggered (Fig 4.20), the non-functional simulation stops before the end of the requested time interval, and a SystemC event is notified, which creates a SystemC instant during which the functional part can react (e.g., by triggering an emergency stop if the temperature is too high).

When one can guarantee that no non-functional event is possible (either because the hardware sensors are only passive components and cannot trigger events, or because we know for sure that the condition will never be met), another strategy is possible: the *functional-ahead*



Request contains:

- Power-state change for each component
- Duration of the interval to simulate
- Conditions on which a non-functional event may be triggered (*halt_condition*).

Response contains:

- Reason for stopping the simulation
- Relevant non-functional values at the end of the interval

Figure 4.18: Cosimulation Interface

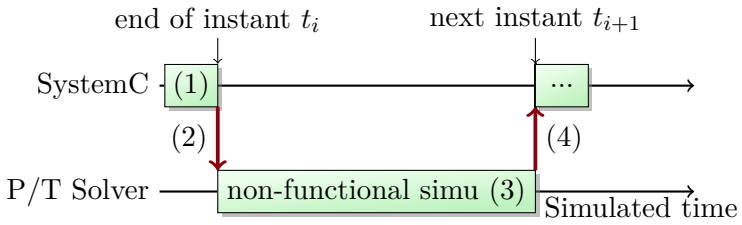


Figure 4.19: *lockstep* cosimulation strategy (for clarity, simulated instants are represented with a non-null width)

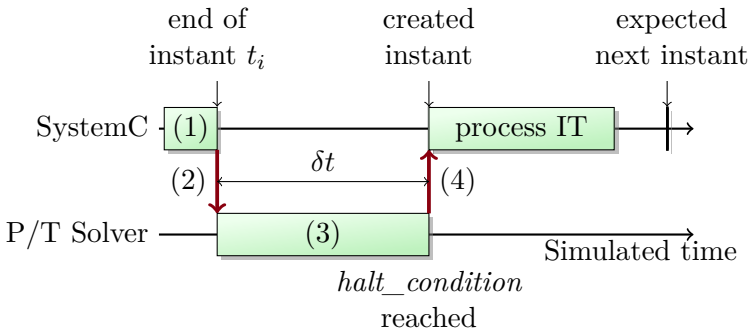


Figure 4.20: *lockstep* cosimulation strategy with interrupt (IT)

strategy, illustrated in Figure 4.21. In this case, the SystemC side runs without launching the non-functional simulation until an access to a sensor is performed. When this happens, a non-functional simulation is requested up to the current SystemC instant. This strategy is less flexible, but requires considerably less request/response round trips, hence can be faster (especially when SystemC and the non-functional solvers run on different machines).

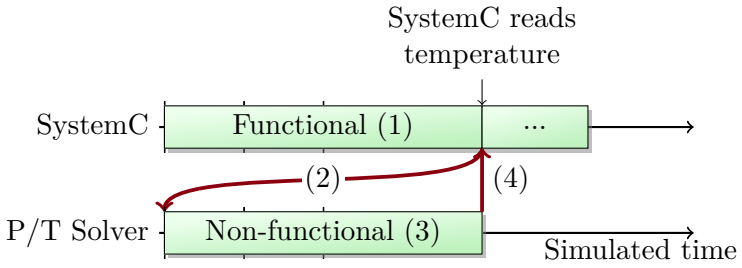


Figure 4.21: *functional ahead* strategy, in the absence of interrupt

These ideas were first experimented in Verimag with Tayeb Bouhadiba using an ATMI-based functional solver. Both the client side (SystemC) and the server-side are implemented in C++. Part of these ideas were then implemented in the industrial tool ACEplorer. The server-side has been rewritten in Java by Docea Power, and the client side was re-written by STMicroelectronics, using their TLM infrastructure and libraries. Thrift acts as an abstraction layer, hence our client can use ACEplorer's server and STMicroelectronics's client can use our non-functional solver without modification. A small case study using the industrial implementation is presented in [CMCM+12].

Traffic Models for Loose Timing

Loose timing and the principles presented in Section 4.1.1 require special attention. An obvious limitation of any power-state based model is that if the timing is too imprecise, then a precise power analysis is not possible (as the power-state model intrinsically needs timing to integrate the power values over time). Still, the timing can be loose and be a reasonable approximation of reality. For an image processing application, for example, a fine-grained model can *compute* how long an image would take to be processed, but a coarse-grained model that considers an image as atomic be calibrated with some timing values close to the actual ones (e.g., considering that decoding one image takes X ms). Calibrated models cannot be very precise, as the actual performance can depend on interactions that cannot be taken into account in calibration (e.g., conflicts on a shared bus). However, using calibrated model is not less precise than using hand-written scenarios, which is a common practice for power/thermal modeling in the industry. This section presents a set of techniques to write approximate power and thermal models based on loosely timed functional models. We cannot recover information which is abstracted away by loose timing, but avoid simulation artifacts that would result from it.

The activity and electrical power-states are modeled the natural way. Consider the example in Figure 4.22.(a). Inserting power-state changes before each computation (Figure 4.22.(b)) actually produces the desired effect: for example, the process will remain in state `computation_f` for 40 time units.

As opposed to that, temperature models for components that require a traffic model do not work naturally on temporally decoupled models, as illustrated in Figure 4.23. The first line shows the transactions that the actual system would perform. The second line shows how these transactions are simulated on a loosely-timed model. In a naive model (4.23.(a)), the dynamic energy consumption of the bus routing these transaction would be modeled at the simulated time

```
f();
wait(40);
commit_f();

wait(event);

g();
wait(35);
commit_g();
```

(a) Example SystemC process

```
set_pwr_state("activity",
              "computation_f");
f();
wait(40);
commit_f();

set_pwr_state("activity",
              "idle");
wait(event);

set_pwr_state("activity",
              "computation_g");
g();
wait(35);
commit_g();
```

(b) Power Instrumented Version of (a)

Figure 4.22: Power Annotation for Loose Timing

when the transactions are executed, hence we would get an instantaneous power consumption at SystemC instants (for clarity, the figure shows cumulative energy instead of power, which would be infinite). The temperature model would therefore compute a peak that does not exist on the real system.

When the temporal decoupling is used with an explicit local clock and on a precisely timed model, the timestamp of the transactions can be used to model the power consumption due to the transaction at the right simulated time. This is the solution used in LIBTLMPWT. However, on loosely timed models, a precise local clock is not always available (in Figure 4.23, the bodies of `f` and `g` may execute behavior in a different order compared to the real system, as explained in Section 4.1.1). In this case, we cannot know for sure when the transactions would be issued by the real system, but we can make the reasonable assumption that they are spread evenly on the time interval. The proposed approach (4.23.(b)) uses this assumption, and turns number of transactions into frequencies (transactions per second) for each initiator. Then, for each SystemC simulation interval, frequencies of all masters are summed, and the result is turned into a power consumption. This way, the modeled temperature is smooth and does not show the unrealistic peaks anymore. If the assumption that transactions are evenly spread is incorrect, then the SystemC programmer can split SystemC instants to refine the model. In the example, `f(); wait(40);` may be turned into e.g., `f1(); wait(15); f2(); wait(35);`. This approach is presented in details in [BMM13a].

Interrupt Management

As explained in Section 4.1.1, a loosely timed model can still be functionally faithful in the presence of interrupts. For example, in Figure 4.22, if an interrupt is received by the component executing `f` at time 20 (in the middle of `f`), then in the real system, the interrupt may abort `f` in the middle of its execution. In the loosely-timed model, `f` has already been completed by the time the interrupt is received, but its functional effect (`commit_f`) has not. It is therefore acceptable to take the interrupt into account at the next SystemC instant (at time 40 in our case).

A natural way to model power consumption in this case would be to check for interrupt after the `wait()` statement, but this would not be faithful: executing the interrupt service routine at the next SystemC instant would consider the power consumption due to this routine at the

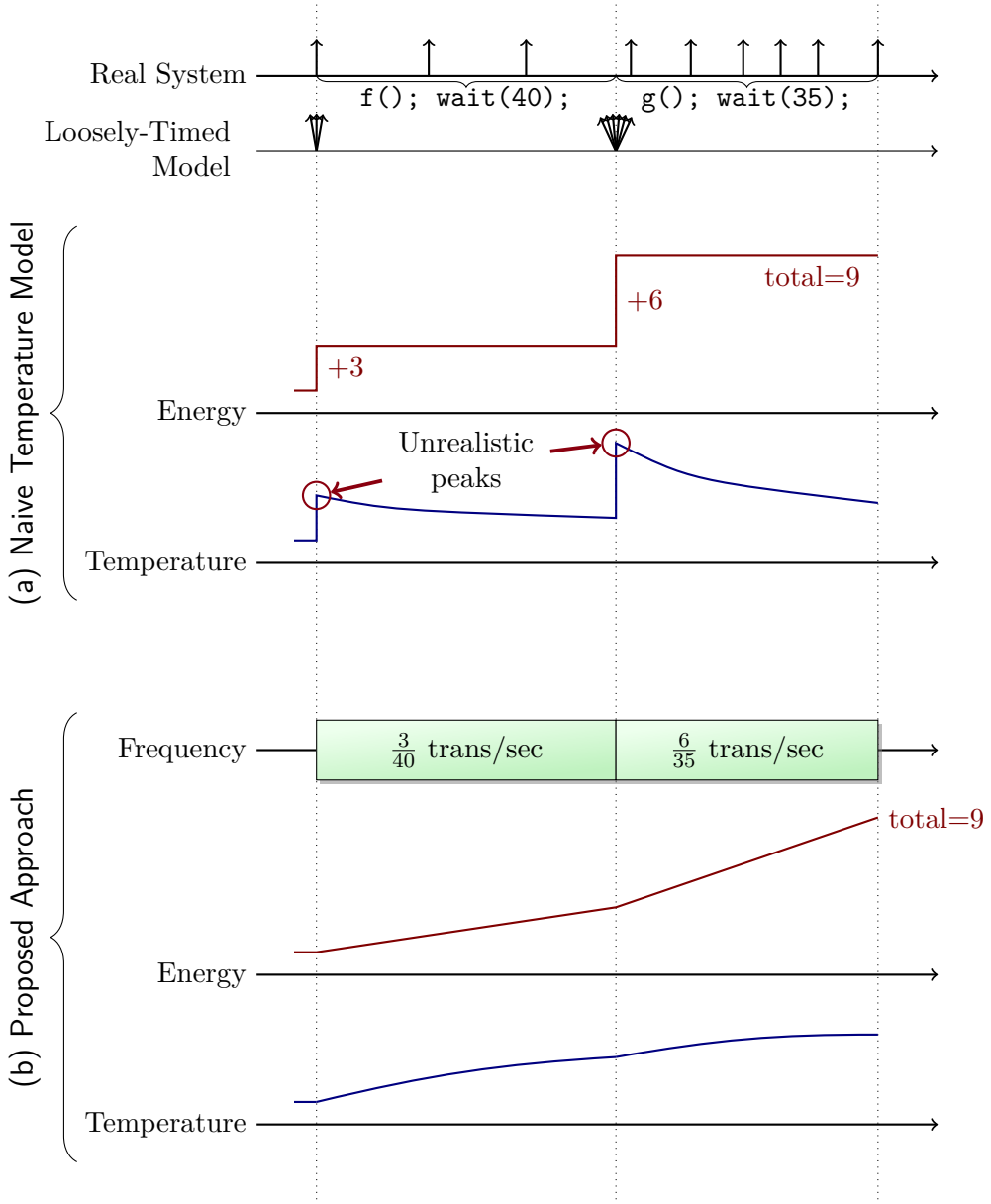


Figure 4.23: Elimination of Simulation Artifacts

wrong point in time. Our approach replaces the call to `wait(time)` between a computation and its functional effect with a `wait(event, time)`, which either completes when enough simulated time has elapsed, or can be interrupted by an event triggered by an interrupt. This way, the interrupt service routine can be executed at the right point it time.

When the interrupt aborts a computation, the non-functional effect of the computation should also be canceled. The power modeling framework presented in [BMM13a] provides primitives for this case, that reset the frequency of transactions for this component. In our example, although `f` has been executed completely, and all transactions it issued are already converted into frequencies on the time interval corresponding to `wait(40)` in the traffic model, the interrupt can reset the transaction frequency for `f` after time 20, hence the non-functional model will actually consider only the first half of `f`.

4.2.5 Validity and Precision of the Models

Obviously, these tools do not allow recovering precision that was lost by raising the level of abstraction. If a component has a complex access pattern to a bus, and this pattern is not modeled, then none of our techniques can accurately model it. This is not surprising, as this would require *guessing* instead of *modeling*. Our contribution is, given a possibly loosely-timed functional platform, to provide modeling tools to create a non-functional model that is as accurate as it can be given the abstraction level of the functional platform. If a better accuracy is needed, then it will require extra modeling work to refine the model.

The validation of these models is a difficult task. Ideally, we should compare the result of the models with a real system, but this is much harder than it seems. Measuring the temperature precisely can be done only on a chip without its packaging, which cannot run at full speed without overheating. This is a non-trivial task for silicon manufacturers, and clearly out of reach for a laboratory like Verimag. Power consumption measurements of individual components would require an instrumented version of the chip (with more pins than the actual system). Instead of validating our models against the real system, we compared several models at several levels of abstraction, considering the lower-level ones as the reference.

Chapter 5

Formal Model of Time: Real-Time Calculus

It is difficult to make predictions, especially about the future.

— various attributions

The previous section dealt with executable models. Although these models are very abstract in comparison to the actual hardware designs, they are still precise enough to be executed, and to execute the actual embedded software. We are now leaving simulation for more abstract models. In this chapter, we are going to abstract away the functionality completely to focus on temporal aspects. The models are far too abstract to execute actual software, but they are well defined and simple enough to allow analytic computations or the use of formal verification tools.

We first recall the basics of modular performance analysis in Section 5.1. Then, we present existing and new approaches to use timed automata within the modular performance analysis framework, in Section 5.2. Section 5.3 presents the tool ac2lus, which allows using Lustre and related tools within the same framework. The proofs performed within ac2lus face the so-called causality problem, which is detailed in Section 5.4, together with an algorithm to solve it. Finally, Section 5.5 concludes on perspectives.

5.1 Modular Performance Analysis and Real-Time Calculus

Modular Performance Analysis (MPA) with Real-Time Calculus (RTC) was briefly presented in Section 1.2.3. In MPA-RTC, a system can be modeled as a set of components that communicate through event streams (see Figure 5.1 for an illustration in the simpler case: one input and one output stream).

Figure 5.1: Concrete Component

In the model, a discrete event carries no data except its date, and has no identity. One can therefore not distinguish events, but only count them. The common way to represent a concrete event stream is to use *cumulative functions* (usually denoted with the letter $\mathcal{R}$), which represent the number of events since the origin of time. Both time and events can be either discrete or

continuous. The illustrations in this document focus on discrete events, which are simpler to represent and to manipulate intuitively, but most of the results also apply to the continuous, or *fluid event* model.

To perform analysis on MPA components, we need to manipulate sets of event streams, i.e., sets of cumulative functions. RTC allows specifying such sets using pairs of *arrival curves* (usually denoted with the letter α), which specify upper (α^u) and lower (α^l) bounds on the number of events that are allowed on time intervals of different sizes. More formally, a cumulative function $\mathcal{R}$ is said to satisfy a pair of arrival curves (α^u, α^l) (denoted by $\mathcal{R} \models (\alpha^u, \alpha^l)$) if and only if

$$\forall x \geq 0, \forall \delta \geq 0, \quad R(x+\delta) - R(x) \in [\alpha^l(\delta), \alpha^u(\delta)]$$

For example, $\alpha^u(3) = 10$ means "there can be at most 10 events in any time window of duration 3".

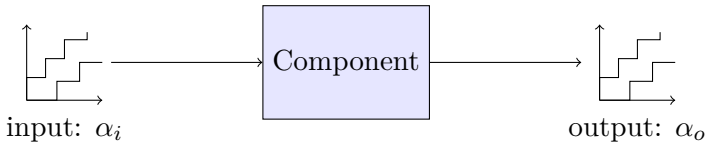


Figure 5.2: Abstract MPA Component

A component in MPA can then be seen as a relation between an input arrival curve and an output arrival curve, as shown in Figure 5.2. Most MPA components are modeled with an input buffer and a computing unit. When an input event arrives, it is either processed or queued in the input buffer. The computing unit consumes events in the input buffer and emits then in the output stream. A common way to model a computing unit is through *service curves*, similar to arrival curves but specifying the amount of computation possible on a time interval. MPA-RTC is an evolution of *network calculus* (where events typically represent data flowing on a network and components represent routers). In RTC, components usually represent real-time tasks, and various scheduling policies can be modeled using service curves appropriately. Based on the arrival and service curves computations, properties like maximal end-to-end delay and maximal buffer size can be computed.

In many cases, components are simple enough to be manipulated in a purely analytic manner. Then, the computations are cheap from the algorithmic point of view, and the results are usually optimal (i.e., they give the tightest possible arrival curves). Unfortunately, the RTC formalism is limited and many components cannot be modeled accurately. In particular, state-based behaviors do not fit in the RTC model. There were attempts to extend RTC to multiple mode systems [PCT08], but none are satisfactory. We started working on RTC with the objective of modeling power-aware components, and could not model properly even naive power management policies like "switch to deep sleep mode when the backlog is empty, and switch back to normal mode when the backlog goes over some threshold".

To work around the limited expressiveness of RTC, connections from RTC to other formalisms like timed automata were developed [LPT09, LPT10, PCTT07, Mok07, PLT11]. The idea of these approaches is to use adapters to translate RTC specification into the equivalent in another language. For example, in Figure 5.3, an input arrival curve α_i is first compiled into a non-deterministic generator which can generate all the streams accepted by α_i and only them. The generated event streams stimulate a model of the component under study, which produces event streams in return. The output adapter is an observer which computes the minimal and maximal numbers of events over multiple time window. For each run, the observer produces upper and lower bounds. A formal verification tool like a model-checker can compute upper and lower bounds for *any* possible run, which provides a valid output arrival curve.

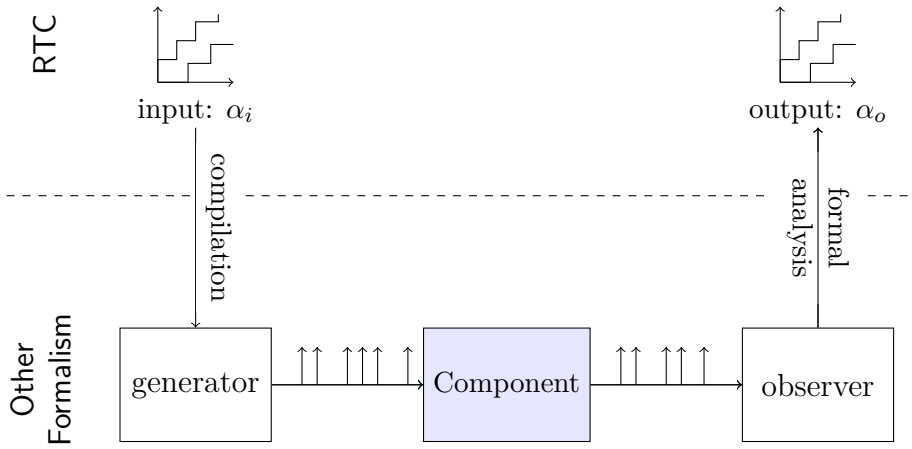


Figure 5.3: Connecting RTC to Other Formalisms

Most approaches connecting RTC to another formalism are variants of this scheme, using different formalisms and different ways to write adapters. The same principle can be used to test real systems: the generator generates actual random stimuli complying with the input curves, and the concrete observer computes the tightest output curves that the output satisfies.

5.2 Modular Performance Analysis and Timed Automata

Connection of RTC to timed automata, using Uppaal [LPY97] as a model-checker, has been well studied in the past [LPT09, LPT10, Mok07, PLT11]. While increasing the expressiveness of MPA, these approaches also suffer from limited scalability. One of the performance issues with Uppaal is that possible values for integer variables are enumerated during model-checking, and the adapters used in the connection use integer variables to count events.

To tackle this problem, we proposed a *granularity*-based abstraction [ALM10] (during the post-doc of Yanhong Liu). Instead of counting individual events, we count packets of N events. As a result, the number of possible values for the counters are decreased by a factor of N. The analysis is then faster and uses less memory, but some information is lost by the abstraction. The analysis can be repeated for several values of the granularity N, and the results of each analysis can be combined and consolidated into a single pair of arrival curves.

The positive conclusion of this experiment is that we could model and analyze a simple example with a power management policy and two states, and get important performance improvements using the granularity abstraction. This conclusion is mitigated by the complexity of the approach, though: the system considered was very simple, and required a lot of manual modeling work. It is not clear whether the approach is general enough to be applicable to non-trivial case-studies.

5.3 Using Lustre as an Intermediate Language in MPA: ac2lus

The other work we did in the same area was to connect RTC to the formal language Lustre. This approach is implemented in the tool ac2lus [AM10a]. It allowed using the abstract interpretation tool Nbac [Jea03] and the bounded model-checker Kind [HT08] (based on SMT-solving), hence brought new formal verification techniques to real-time calculus.

The tool ac2lus uses a discrete-time model: one synchronous clock tick corresponds to one

time unit. Event streams are modeled with integer Lustre variables: the value of the variable at each clock tick represents the number of events received during this instant.

Using Lustre as a modeling language and the verification tools Kind and Nbac is interesting in two ways. First, abstract interpretation and SMT based model-checking perform differently from timed-automata model-checkers, hence can solve different classes of problems. As discussed above, timed-automata based techniques deal with event counters in an enumerative manner, while SMT and abstract interpretation work with them symbolically, hence have no problem with large values. As a result, our approach scales better than the existing ones when buffer sizes are large for example. On the other hand, timed automata work in dense time and model-checkers deal with it symbolically, while our use of discrete time implies one step of the model checker for each time step. Our approach therefore scales badly when time constants of the system are big in comparison to the time step used in the model.

Also, our experience is that modeling a component using the Lustre language was much easier than using Uppaal's timed automata, which are both powerful and error-prone. We had several discussions about complex timed automata products, whether adding a transition could add behaviors or would enforce synchronization, what could happen in cases of deadlocks... which we didn't need when using Lustre. The Lustre language is not very expressive, but is also very simple and makes it easy to have a clean, possibly hierarchical structure. The simplicity of the language avoids easy misinterpretation of its semantics, and the clean structure allows managing non-trivial pieces of code (we use Lustre as a modeling language here, but the same language has already been used successfully to implement systems orders of magnitude more complex at e.g., Airbus).

To illustrate our approach, a model of a greedy processing component (which processes events when it has resource, and queue them otherwise) is shown in Figure 5.4. At each clock tick, it receives a number of input events on `in_seq`, and returns the number of output events in `out_seq`. The number of events stored in the internal buffer is kept in the local variable `backlog`. The model is written manually, and the tool ac2lus will use verification tools to perform some computations with it.

Ac2lus can use the generator+observer scheme of Figure 5.3, but can also use observers for both the input and output adapters. For a given pair of arrival curves, ac2lus can generate a Lustre observer. The observer takes an event stream as input, and returns a Boolean saying whether the stream satisfies the arrival curves up to the current instant. An example observer is shown in Figure 5.5. Variables `c1`, `c2` and `c3` count the number of events for windows of time of durations 1, 2 and 3 respectively. We then check whether these values are greater than α^u (0, 0, 1) and smaller than α^l (2, 3, 5). Figure 5.5 is simplified for the example: the real generated code has to special-case the first two instants, where the values of `c2` and `c3` are not relevant.

Given an input arrival curve and a component model, we can compute the output curve as follows:

1. We generate an observer for the input arrival curve. The component receives a non-deterministic, arbitrary input stream, which is also given as input to the observer. The observer provides a Boolean `in_ok` that tells whether the input satisfies the arrival curve.

2. A candidate output arrival curve is chosen (typically, a pair of curve with only one point), and an output observer is generated for it. It provides a Boolean `out_ok`.

3. We check whether the property `(in_ok => out_ok)` is provable with Kind or Nbac. If so, the candidate arrival curve is valid. We can keep the constraint, and try a tighter one. If the property is not provable, then we can try with a looser constraint. Ac2lus uses a binary search for each point of each output curve.

```
node gpc (in_seq: int; in_res: int)
 returns (out_seq: int; out_res: int)
var
  backlog: int; work: int;
  empty_queue: bool;
let
  -- events to compute at the current
  -- instant (accumulated + new work)
  work = in_seq -> (in_seq + pre(backlog));

  -- whether we'll empty the queue at the
  -- current instant
  empty_queue = (work <= in_res);

  -- amount of work accumulated in the past
  backlog = if (empty_queue) then 0
            else work - out_seq;

  -- events produced
  out_seq = if (empty_queue) then work
            else in_res;

  -- resource remaining after running
  out_res = in_res - out_seq;
tel
```

Figure 5.4: Greedy Processing Component in Lustre

Point 3 requires a bit of attention. What we want to prove is that if the input pair of curves is valid, then so is the output pair of curves. Formally, this is $(R_i \models \alpha_i) \Rightarrow (R_o \models \alpha_o)$, with R_i being an arbitrary input stream, and R_o the output stream of the component when receiving R_i. This would be `G(in_ok) => G(out_ok)` if we could express properties using the `G` (always) LTL [Pnu77] operator. The property we can express in Lustre actually means `G(in_ok => out_ok)`, which is stronger. In other words, it is possible that `(in_ok => out_ok)` be false even though the candidate output curve is correct. This happens if the input stream has not *yet* violated its arrival curve (i.e., `in_ok` is still true), but will inevitably do so in the future, and `out_ok` is false. We will get back to this issue in more details in Section 5.4.1, and provide a solution called the *causality closure*.

Although the main use of ac2lus is to compute output curves as function of input curves, the tool actually allows proving arbitrary properties on the model. Using the same binary search scheme, we can compute the best provable bound on any value directly (like maximal buffer size).

Ac2lus' performances could be improved in two ways. First, as already mentioned, the use of discrete time requires many iteration of the proof engine. Acceleration techniques, like the ones implemented in the ASPIC [GH06] tool could probably reduce the complexity of the problem. Then, the binary search scheme requires launching the proof engines (Kind and Nbac) many times for each computation. We do not consider this as a very serious issue, since the number of times we launch proof engines is linear in the number of points to compute, and logarithmic in the orders of magnitude of each point, while the computation performed by the tool itself can be exponential. Still, it would be interesting to use a tool that outputs an invariant on the relevant numerical variables instead of having to perform a binary search.

```
node ok_points (sequence: int)
       returns (ok: bool)
var
  c1, c2, c3: int;
let
  c1 = sequence;
  c2 = sequence -> pre(c1) + sequence;
  c3 = sequence -> pre(c2) + sequence;
  ok = (0 <= c1 and c1 <= 2) and
       (0 <= c2 and c2 <= 3) and
       (1 <= c3 and c3 <= 5);
tel
```

Figure 5.5: Simplified Lustre Code of a Deterministic Observer

5.4 The Causality Problem

The previous section exhibited a *causality* problem. The problem occurs when a finite event stream satisfies a pair of arrival curve α, and no infinite extension of this finite stream can satisfy α. Intuitively, the finite prefix *seems* valid with respect to the arrival curves, but it is not a prefix on any infinite stream: it will inevitably violate a constraint in the future, hence cannot be considered as valid. In other words, there are constraints that are not explicit on the arrival curves, but that an event stream has to satisfy to be valid. We call these constraints *implicit constraints*.

5.4.1 Defining Causality

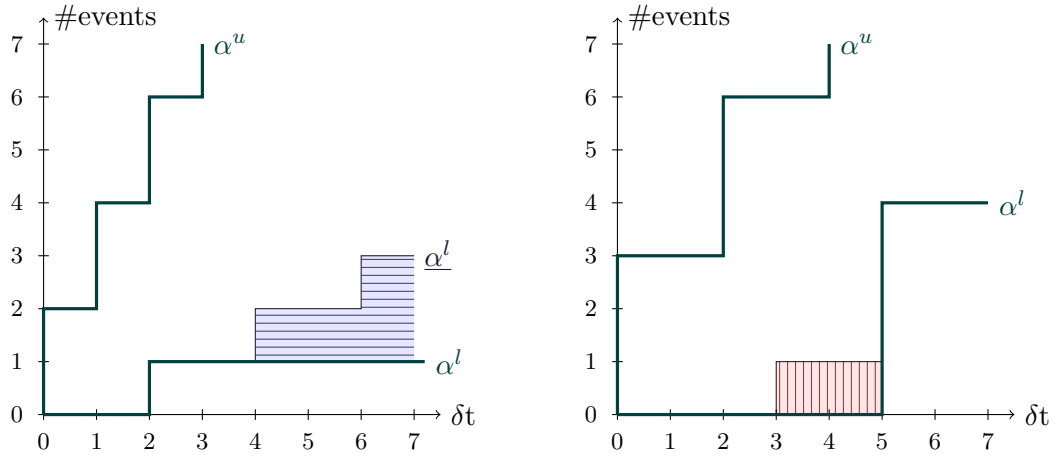
Consider the pair of arrival curve in Figure 5.6.(a). The curve α^l exhibits a well-known [LBT01] kind of implicit constraint: the curve does not explicitly forbid to emit a single event in a window of time of duration 5 ($\alpha^l(6) = 1$), but since the curve imposes at least one event every 2 time units, it is clear that a valid event stream has to emit at least 2 event during any interval of time of size 5 time units. Technically, we say that α^l is not *super-additive*, and it is easy to build the *super-additive closure* $\underline{\alpha^l}$ of α^l, which is equivalent to α^l and makes this constraint explicit. We call the area between $\underline{\alpha^l}$ and α^l *unreachable regions* (blue, vertically hatched area in Figure 5.6.(a)): no cumulative curve R with $R(0) = 0$ can enter these regions without violating α^l.

The case of interest in this chapter is the one of Figure 5.6.(b), which surprisingly received very little attention from the RTC community. To understand the issue, consider an event stream that emits no event for 4 time units. Doing so does not violate any constraint, but $\alpha^l(5) = 4$ implies that at least 4 events will have to be emitted during the next instant. This conflicts with $\alpha^u(1) = 3$ which does not allow more than 3 events at the same instant. Intuitively, we say that the event stream entered a *forbidden region* (red, horizontally hatched area in Figure 5.6.(b)).

Pairs of arrival curves for which this situation cannot occur are called *causal*. In other words, causal pairs of arrival curves are curves for which any event stream that verifies the curves *up to* some point in time can be extended into an infinite event stream that verifies the curves. We provided a formal definition in [AM10b]. A non-causal pair of curves is either unsatisfiable or equivalent to a strictly tighter pair of curves called the *causality closure*, which is causal.

Non-causal pairs of curves are problematic with several respects:

- Obviously, because they contain implicit constraints, they are not optimal. When a computation yields a non-causal pair of curves, applying the causality closure on it gives tighter bounds for the same model. Indeed, we showed in [AM10b] how to apply the causality



(a) "Unreachable" regions (b) "Forbidden" regions

Figure 5.6: Implicit Constraints on Arrival Curves

closure to obtain the tightest possible pair of arrival curves equivalent to a given pair of curves.

- In a verification context, non-causal pairs of curves at the input of a component may yield spurious counter-examples, as is the case with ac2lus. Using causal pairs of curves makes the problem both easier to think with for human being, and simpler to verify for automatic tools.

- When generating concrete event stream based on an arrival curve (for example, to test a concrete implementation of a component), a common approach is, at each point in time, to compute the lower and upper bounds on the number of events that can be emitted. Then, the number of event to emit at the current instant is chosen within the interval. Such generator applied on Figure 5.6.(b) could emit no event for 4 time units, and then encounter a deadlock, as explained above. A deadlock would on the other hand be impossible with causal pairs of curves.

5.4.2 Solving the Causality Problem: The Causality Closure

We will now define the causality closure (temporarily denoted by $(\alpha^{u*}, \alpha^{l*})$) of a pair of curves (α^u, α^l). A necessary condition to be able to extend a prefix of an event stream after time t is that the constraint imposed by α^u on intervals $[t, t+\delta]$ do not conflict with the ones imposed by α^l on $[0, t+\delta]$. The smallest value of α^{l*} for which this is true is $\inf_{\delta \geq 0} \left\{ \alpha^l(t+\delta) - \alpha^u(\delta) \right\} \stackrel{\text{def}}{=} \left(\alpha^l \oslash \alpha^u \right)(t)$. Similarly, the largest of α^{u*} value for which the constraints of α^l on $[t, t+\delta]$ will not conflict with α^u on $[0, t+\delta]$ is $\sup_{\delta \geq 0} \left\{ \alpha^l(t+\delta) - \alpha^u(\delta) \right\} \stackrel{\text{def}}{=} \left(\alpha^u \overline{\oslash} \alpha^l \right)(t)$. $\oslash$ is the *min-plus deconvolution* operator, and $\overline{\oslash}$ is the *max-plus deconvolution*.

A first intuition was then to define the causality closure of (α^u, α^l) as $\left(\alpha^u \overline{\oslash} \alpha^l, \alpha^l \oslash \alpha^u \right)$. This is unfortunately not sufficient, as shown in Figure 5.7. Applying the deconvolution operators once yields $\alpha^{l*}(4) = 1$, which is actually forbidden since the new upper curve $\alpha^u \overline{\oslash} \alpha^l$ prevents emitting more than 2 events at the next instant. In this case, this could be fixed by applying the super-additive closure to $\alpha^l \oslash \alpha^u$, but this is not sufficient in the general case.

A second attempt was to iterate the convolution operators. Indeed, when repeating the deconvolution operators of Figure 5.7, one obtains the curves in Figure 5.8.(a). In this case, we

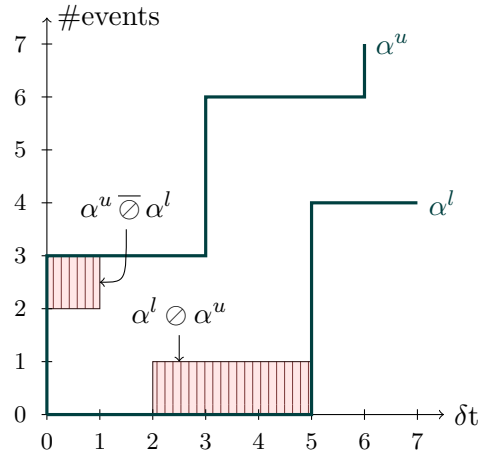
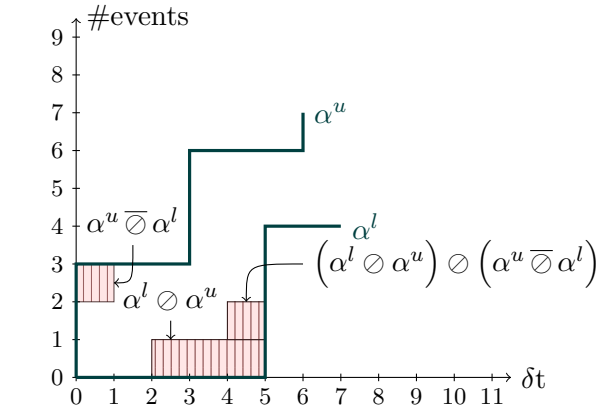
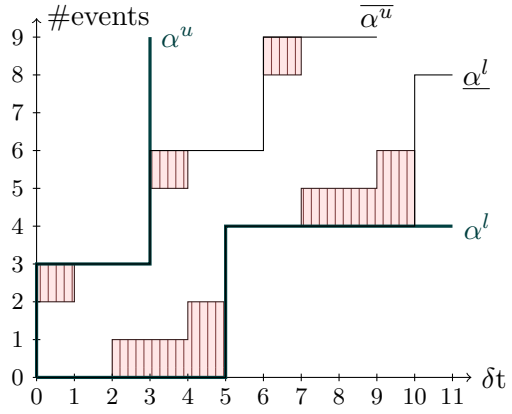


Figure 5.7: Applying the Deconvolution Operators



(a) Iterating the Deconvolution Operators



(b) Deconvolution Operators on Sub/Super-Additive Curves

Figure 5.8: Iterating the Deconvolution Operators

get the causality closure after two iterations. We showed in [AM10b] that a pair of curves stable by deconvolution was causal, hence, if the iterations reach a fixpoint, then this fixpoint is causal. The termination of the fixpoint iteration usually terminates after a few steps in practice, but whether the termination is guaranteed in the general case (continuous time and fluid event) is an open question.

The approach we propose is not to iterate the deconvolution operators, but to apply the sub-additive and super-additive closure to the curves to remove unreachable regions *before* applying the deconvolutions. This is illustrated in Figure 5.8.(b). When the input curves are sub-additive and super-additive, we showed in [AM10b] that the application of deconvolution operators reaches the fixpoint at the first iteration. In other words, the pair of curves $\left(\overline{\alpha^u} \overline{\oslash} \underline{\alpha^l}, \underline{\alpha^l} \oslash \overline{\alpha^u}\right)$ is causal (unless it is trivially unsatisfiable, i.e., if the curves cross at some point). We call it the *causality closure* of (α^u, α^l).

Unless trivially unsatisfiable, $\left(\overline{\alpha^u} \overline{\oslash} \underline{\alpha^l}, \underline{\alpha^l} \oslash \overline{\alpha^u}\right)$ also has several interesting properties:

1. It is sub/super-additive (i.e., has no unreachable regions)
2. It is the tightest pair of curves equivalent to (α^u, α^l)

Conversely, if (α^u, α^l) is the tightest pair of curves defining a set of event streams, then it is also causal and sub/super-additive.

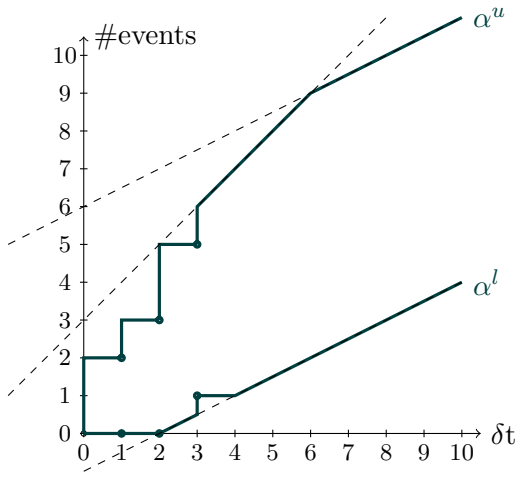


Figure 5.9: Example $\mathcal{U}pac$ Curve: α^u contains 3 explicit points and two affine pieces, and α^l 3 points and 1 affine piece

This last result probably explains why the problem hasn't been studied in the past: any optimal computation yields causal curves, and the formula used in RTC are usually optimal. On the other hand, connections of RTC to other formalisms have to perform some conservative abstractions (either because the proof engine does some abstraction, or because it may timeout). As a result, non-causal curves may be computed as the output of a component, and may be problematic when used as the input of the next component. The causality closure solves this issue.

In the abstract interpretation framework, a pair of arrival curves can be seen as an abstraction of a set of streams. The abstraction function (α) returns the tightest pair of curves that a set of streams satisfies. The concretization function (γ) gives the set of streams satisfied by a given pair of curves. The causality closure corresponds to the $\alpha \circ \gamma$ operation, which refines an abstract value without changing its meaning.

5.4.3 Implementation in ac2lus for $\mathcal{U}pac$ Curves

The algorithms and theorems presented in [AM10b] deal with the most general case, and are applicable on any class of curves where the usual convolution and deconvolution operators in min/max-plus algebra are defined. The tool ac2lus works with a class of arrival curves called $\mathcal{U}pac$ (*ultimately piecewise-affine curves*), where a curve contains a finite set of points, plus a finite set of affine pieces (see for example Figure 5.9).

The $\mathcal{U}pac$ class of curves is unfortunately not closed under convolution and deconvolution operators, hence the approach requires some adaptations. The causality closure for the $\mathcal{U}pac$ class of curves is presented in detail in [AM11], and have been partially implemented in ac2lus by Guillaume Sarrazin.

The first step of our causality closure algorithm for $\mathcal{U}pac$ curves is a normalization. In the general case, it consists in:

1. Making sure that the explicit points of α^u (resp. α^l) are below (resp. above) the affine pieces.
2. Removing irrelevant affine pieces, i.e., pieces that are weaker than the constraints due to explicit points.
3. Add explicit points up to the abscissa where the affine pieces dominate
4. Apply the sub-additive closure to α^u and the super-additive closure to α^l.

There are particular cases described in [AM11] if α^u and/or α^l has no relevant affine pieces. Then, a bounded form of the deconvolution operators can be applied to get the causality closure.

5.5 Towards a more General Modular Performance Analysis Framework

Our work on modular performance analysis with real-time calculus provides both theoretical foundations and tools to connect RTC to other formalisms like timed automata or Lustre. This allows modeling some individual components with formalisms more expressive than RTC, and as a result provides more precise results (it is always possible to model a system in RTC, by defining $(\alpha^u, \alpha^l) = (+\infty, 0)$, but it is not always possible to model it precisely).

In a system with several components modeled using a state-based formalism, one can either perform a global analysis on the product of the automata, or a local analysis for each component, computing output arrival curves for each of them. The first option is limited in scalability, as the complexity is exponential in the number of elements of the product. In the case of local analysis, the complexity grows linearly in the number of component, but the use of arrival curves at the interface of components loses precision: if the component itself needs a state-based formalism to be accurately modeled, then the output stream may also need such formalism to be described. Going back from the state-based universe to the RTC universe can be seen as a projection that loses information in this case.

Our long-term goal would be to allow using a state-based formalism not only as a modeling tool to describe components, but also in the interface of components. Experiments were started in this direction with Xavier Jean.

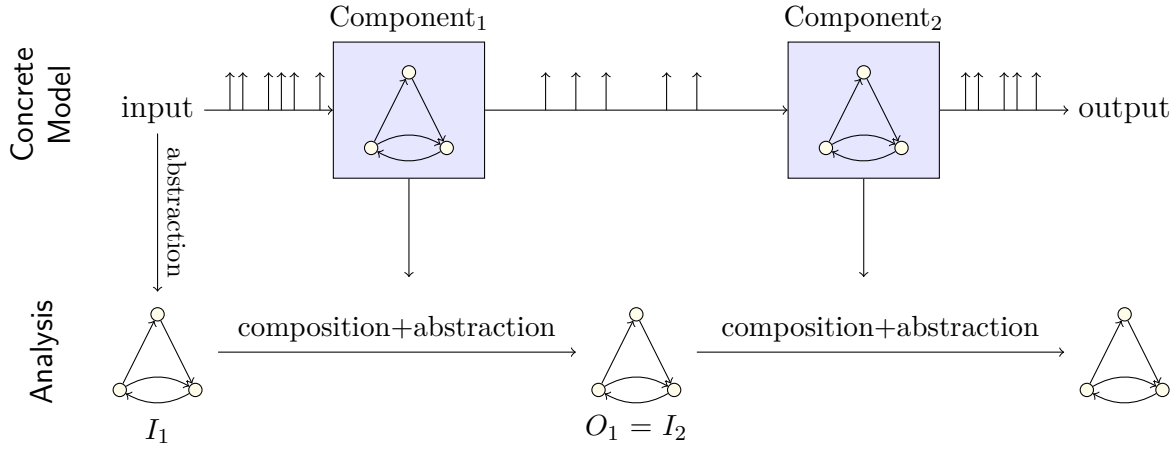


Figure 5.10: Modular Performance Analysis with State-Based Formalisms

The overall idea is illustrated in Figure 5.10. The components themselves are modeled with automata, which act as transducers. The event-streams are modeled with automata which are generators of the possible concrete event-streams.

Computing the automaton O_i for the output of a component C_i given its input I_i requires two operations:

1. A composition between I_i and C_i, which computes an automaton O'_i that precisely describes the output stream.
2. An abstraction operator that turns O'_i into a simpler and possibly less precise automaton O_i.

In the experiments performed by Xavier Jean, automata contain both explicit control points and numerical variables (for example, to count events). The composition between the input and the component is the usual automaton product. The abstract interpretation tool Nbac [Jea03] is used to find an invariant on the variables of the product, and the result is projected on the control points and variables of one of the automata I_i or C_i, with the constraints found by Nbac added to transition guards. A preliminary implementation of the principle was written, but was never completed by lack of manpower.

Similar attempts have also been performed in the past, like [BSBM09] which works with timed automata and uses projection on a subset of clocks as abstraction operator.

Chapter 6

Conclusion

Your job is being a professor and researcher: That's one hell of a good excuse for some of the brain-damages of Minix.

— Linus Torvalds to Tanenbaum, *famous Usenet flame-war thread (1992)*

6.1 Summary and General Principles

In summary, my research work during the last decade was dealing with high-level modeling and verification of embedded systems. New techniques to model functional and non-functional properties, and new tools to evaluate or formally verify them have been proposed. The models range from precise and executable models for Systems-on-a-Chip, typically written in SystemC, to purely analytical models for performance analysis. We also worked on formal verification of sequential software.

One general principle through my past and future research is to use concrete problems, as much as possible coming from the industry, as a guide. Not all my research is directly applicable in practice, but the medium or long-term direction is always a concrete problem. In other words, the privileged approach is to start from practical issues, to extract theoretical (and hopefully elegant and interesting!) questions out of it. Being guided by concrete issues does not imply doing research only for immediate needs of the industry.

One noticeable instance of industrial partnership is the relationship between STMicroelectronics and Verimag, since 2002. We had the opportunity to work with a team that is both very close to production and interested in research. This fruitful collaboration led us through a lot of different practical issues, and gave us a point of view that we couldn't have by looking only at the state of the art publications. It also allowed me to keep the course I give on the subject in Ensimag as close to industrial reality as possible.

I often heard critics on SystemC-related research, from people considering that Science should be about developing new tools and languages, as opposed to dealing with existing (imperfect) ones. I do not consider existing and non-formal language as being incompatible with interesting science. Indeed, starting from down-to-earth issues like "How to verify the correctness of a SystemC program?" brought us to a lot of different questions like "How to combine static analysis and runtime information?" and led us through a many different approaches using e.g., different kinds of automata.

One very nice tool to achieve industrial partnership is the *CIFRE*[1] Ph.D. I did a CIFRE Ph.D myself (STMicroelectronics/Verimag), co-supervised the CIFRE Ph.D of Giovanni Fun-

[1] Conventions Industrielles de Formation par la REcherche, a french facility where a Ph.D student is hired by a company and supervised by a public laboratory

chal, and worked with several other CIFRE (or other co-funded thesis) students. In all cases, the framework was a very nice way to create links between the company and the laboratory. The dual supervision of the student creates links between industrial and academic supervisors. The student has access to confidential data on the industrial side, and benefits from the supervision of the public laboratory. The approaches developed during the Ph.D I've been involved with did not produce production-ready tools, but did produce ideas that were then reused in tools maintained by engineers after the student left (or was hired by the company, in the case of Jérôme Cornet!). I hope to find working contexts similar to these in the future (either using CIFRE or Ph.D within collaborative projects).

I kept an interaction between teaching and research. My research on transaction-level modeling is directly useful to the course I give in the Ensimag on the same subject. I first encountered topics like energy modeling or parallelism through research, and could introduce the outcome of research a few months after in the Ensimag course. The opposite is true also: the example platform developed as part of an Ensimag lab-work was a nice example to experiment research ideas. When a pedagogical innovation is interesting enough, it deserves to be shared with publications like other research projects [Moy11a, Moy11b].

I like research and theoretical issues, but I also like coding and practical results. Except [MMK10], [MMC+08] and [BGM+09b], all the papers in my publication list correspond to a tool or a feature of a tool to which I contributed directly (i.e., writing code, although not always as a main author). I think implementing algorithms gives a tangible way to evaluate theoretical results, and helps finding relevant research directions. For example, there is no good theoretical way to compare two abstract interpretation algorithms (because of the non-monotonicity of widening operators, being more precise at a stage of the analysis can lead to less precise global results). Empirical comparison based on an implementation (e.g., using PAGAI), on the other hand, can tell which algorithm performs better *in practice*, and identify weaknesses in existing algorithms, hence new problems to be solved by future research.

Tangible results are also a great tool to supervise students. The last few years have seen a great deal of interest in agile methods like Scrum, and I've worked a lot with them in teaching. While many agile practices are hardly relevant in a research context, at least the notions of iterations, user-stories and "definition of done" can actually fit relatively well. One agile principle is to have at the end of every iterations (two to four weeks) a *demonstrable* product. In industry, "demonstrable" generally means "ready to be released to the client", which usually does not apply in research. On the other hand, forcing ourselves to have a "demonstrable" tool, in the sense that the tool gives some tangible results is a way to avoid the most common pitfalls like "the tool is 80% done although right now nothing really works" (which is too often claimed during 80% of the lifetime of the project) or dispersion of efforts among any projects in parallel, without actually completing many of them. Worse, I had several experiences supervising students that were actually working on dead pieces of code: trying to compare a reference implementation with an optimized one, we were actually comparing the reference implementation with itself! In both cases, doing a first iteration with a user story as simple as "if I add a `printf` to the optimized code, then I should see it in the differences between the optimized and the reference version" would have exhibited the problem trivially, and much earlier than we actually did.

6.2 Future Directions

My general research topic will continue to be high-level modeling of embedded systems, with a particular emphasis on transaction-level modeling.

6.2.1 Performance Optimization for SystemC Simulations

There are at least two hot topics in TLM today: making faster models, and modeling more properties of the real system. Simulation speed of TLM is already orders of magnitude faster than lower-level ones, but still reaching its limits (when the system being simulated is a 4k UHD video decoder in a television with a 100Hz refresh rate, a sequential and purely software implementation will necessarily be slow ...).

Continuing the work on optimized compilation of SystemC (see the work on Tweto, Section 3.3) should allow reducing the overhead of communications in a TLM simulation, or reduce the manual effort to optimize it (compared to using DMI). Also, as already demonstrated by the tool Scoot [BKS08], implementing a SystemC-specific compiler can allow some optimization of the scheduler (essentially replacing context-switches with `goto` statements when possible), and could probably be used to allow a better parallelization (as proposed by [Bou07]).

Optimized compilation alone would be interesting to have, but the biggest performance bottleneck is clearly the fact that simulations are sequential. It is particularly frustrating since the system being simulated is itself highly parallel, and almost any host machine today is multi-core. There are multiple challenges here, as the solutions should not only allow high-performance parallelism, but also deal with legacy code. The ideas presented in Section 4.1.5 already provide one solution, but there is still a lot to be done. On the one hand, we must profile and optimize the low-level implementation of the library (e.g using atomic instructions instead of blocking system calls, load balancing ...). On the other, we should provide more tools to the user to describe the platform at a high level of abstraction, and take advantage of the new constructs for better parallelization. Parallelization should be dealt with early, before the low-level implementation forced sequentiality. The OpenES European project will allow us to continue working on the subject, and apply it to real case-studies. Hopefully, we will also be able to discuss with the SystemC language working group and help introducing more tools for parallelism in the SystemC standard.

6.2.2 Thermal Modeling for Systems-on-a-Chip

Other challenging issues are the modeling of non-functional properties at the transaction-level. We already started some power and thermal modeling at different timing granularities, but there are still many areas for future developments. Currently, our models are based on power-state, and the functional simulation drives the power-state machine. It would be interesting to express more properties within the non-functional model, like the time taken by transitions, or constraints on the time spent in some particular states. The same information could be used in purely non-functional simulations (based on scenarios) and in functional/non-functional cosimulation. We would also like to introduce non-determinism in non-functional models as a way to model uncertainty and to ensure robustness, and allow hierarchical models of non-functional properties.

6.2.3 Formal Verification of SystemC Programs

Formal verification of SystemC is still an interesting problem too, and is still unsolved for large case-studies. I have little hope that anyone will ever be able to formally verify a complete TLM platform automatically, but research on formal verification still brings actual direct and indirect contributions to the state of the art. As a researcher, I think formal verification is an excellent intellectual exercise. It enforces rigor and formalization; it forces us to think exhaustively, in terms of sets of possible behaviors and not only in terms of individual executions. The fact that my first contact with SystemC was through formal verification influenced a lot the way I dealt with issues other than verification afterwards. From a more concrete point of view,

formal verification tools can be used to verify parts of a system, and tools that extract a formal model from SystemC modules can be used to extract abstractions of these modules (see e.g., Section 3.2.4). For compositional approaches to work, we need both to have tools able to deal with reasonably large subsystems, and to have ways to compose these proofs.

Past works provided the basis for several interesting verification approaches. The work done on translation of C++ into Signal showed promising performances, and the tool PinaVM provides the technical framework to extend this to actual SystemC code. It would be interesting to write a PinaVM backend that would generate synchronous code taking advantage of the SSA form. PinaVM was originally written to connect to the tool ConcurInterproc by generating code in the Simple language, but did not achieve this goal yet. Completing the Simple backend would allow performing clever interprocedural analysis on SystemC code.

6.2.4 Modular Performance Analysis: Beyond Real-Time Calculus?

The work carried out on Modular Performance Analysis with Real-Time Calculus (RTC) consolidated and extended the existing works on the connection of RTC to some state-based formalisms. The next step would be to allow replacing not only the RTC modules, but also the arrival curves used to describe the *interface* between RTC modules with some state-based formalism. We started some preliminary work to compute a non-deterministic state-machine as the output of the performance analysis of a component, but do not have yet a solid approach and a case-study.

6.2.5 Abstract Interpretation

Some very interesting work on abstract interpretation was started with the tool PAGAI. Current work focuses on compositional analysis, considering pieces of code (e.g., nested loop or function calls) first as non-deterministic black boxes, and then refining them on demand, guided by a property. This approach is very promising and should open the door to many new techniques in the near future. We will need new heuristics to know when and how to refine a block, how to re-use a previous analysis with a different precondition, ... PAGAI is already able to run on real, large programs, so once the compositional analysis will start working, we will be able to benchmark it and to improve it for real use-cases.

6.2.6 Implementation of Critical Systems on Multi-Core Architectures

Critical systems were the main topic around which the Verimag laboratory was originally founded. Model-checking and abstract interpretation allowed verifying formally their correctness, and the Lustre synchronous language was successfully transferred in the industry through the SCADE tool. At that time, critical embedded software could run on simple hardware processors (e.g., Motorola 68000). These single-core processors are no longer sufficient in many cases. Unfortunately, general-purpose multi-core processors are not suited for critical systems, as they use too many optimizations that make them unpredictable, in particular in their timing aspects.

We started investigating the area of multi-, or many-core for critical systems. This needs an overall view of hardware and software, to build a programming model that minimizes interactions between tasks and allows formal analysis of functional and non-functional properties.

Chapter 7

References

7.1 Bibliography

[ALM10] Karine Altisen, Yanhong Liu, and Matthieu Moy. Performance evaluation of components using a granularity-based interface between real-time calculus and timed automata. In *Eighth Workshop on Quantitative Aspects of Programming Languages (QAPL)*, Paphos, Cyprus, March 2010. Electronic Proceedings in Theoretical Computer Science. Cited in sections 1.2.3 and 5.2.

[AM10a] Karine Altisen and Matthieu Moy. ac2lus: Bringing SMT-solving and abstract interpretation techniques to real-time calculus through the synchronous language Lustre. In *22nd Euromicro Conference on Real-Time Systems (ECRTS)*, Brussels, Belgium, July 2010. Acceptance rate: 27/112 = 24%. Cited in sections 1.2.3 and 5.3.

[AM10b] Karine Altisen and Matthieu Moy. Arrival curves for real-time calculus: the causality problem and its solutions. In J. Esparza and R. Majumdar, editors, *TACAS*, pages 358–372, March 2010. Acceptance rate: 24/110 = 21%. Cited in sections 1.2.3, 5.4.1, 5.4.2, 5.4.2, and 5.4.3.

[AM11] Karine Altisen and Matthieu Moy. Causality closure for a new class of curves in real-time calculus. In *Proceedings of the 1st International Workshop on Worst-Case Traversal Time*, pages 3–10, Vienna, Autriche, 2011. ACM. Cited in sections 1.2.3, 5.4.3, and 5.4.3.

[AMD+94] Mazhar Alidina, José Monteiro, Srinivas Devadas, Abhijit Ghosh, and Marios Papaefthymiou. Precomputation-based sequential logic optimization for low power. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2(4):426–436, 1994. Cited in section 4.2.1.

[AS13] Gianluca Amato and Francesca Scozzari. Localizing widening and narrowing. In Francesco Logozzo and Manuel Fähndrich, editors, *SAS*, volume 7935 of *Lecture Notes in Computer Science*, pages 25–42. Springer, 2013. Cited in section 2.4.3.

[BCH+85] J-L. Bergerand, P. Caspi, N. Halbwachs, D. Pilaud, and E. Pilaud. Outline of a real time data-flow language. In *RTSS*, 1985. Cited in section 1.1.4.

[BCTB13] A. Bartolini, M. Cacciari, A. Tilli, and L. Benini. Thermal and energy management of high-performance multicores: Distributed and self-calibrating model-predictive controller. *Parallel and Distributed Systems, IEEE Transactions on*, 24(1):170–183, 2013. Cited in section 4.2.1.

[BDM96] Luca Benini and Giovanni De Micheli. Automatic synthesis of low-power gated-clock finite-state machines. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 15(6):630–643, 1996. Cited in section 4.2.1.

[Ber89] Gérard Berry. Real time programming: special purpose or general purpose languages. Rapport de recherche RR-1065, INRIA, 1989. Cited in section 2.

[Ber00] Gérard Berry. The Esterel v5 language primer - version v5_91, 2000. Cited in section 4.1.5.

[BFH+01] G. Behrmann, A. Fehnker, T. Hune, K. Larsen, P. Pettersson, J. Romijn, and F. Vaandrager. Minimum-cost reachability for priced time automata. *Hybrid Systems: Computation and Control*, 2001. Cited in section 4.2.2.

[BGM+09a] Loïc Besnard, Thierry Gautier, Florence Maraninchi, Matthieu Moy, and Jean-Pierre Talpin. Comparative study of approaches to semantics extraction and virtual prototyping of system-level models. Technical report, Verimag and IRISA, 2009. Cited in section 3.2.3.

[BGM+09b] Loïc Besnard, Thierry Gautier, Matthieu Moy, Jean-Pierre Talpin, Kenneth Johnson, and Florence Maraninchi. Automatic translation of C/C++ parallel code into synchronous formalism using an SSA intermediate form. In *Ninth International Workshop on Automated Verification of Critical Systems (AVOCS'09)*. Electronic Communications of the EASST, September 2009. Cited in sections 1.2.2, 3.1.4, 3.2.3, and 6.1.

[BHS98] Luca Benini, Robin Hodgson, and Polly Siegel. System-level power estimation and optimization. In *Proceedings of the 1998 international symposium on Low power electronics and design*, ISLPED '98, pages 173–178, New York, NY, USA, 1998. ACM. Cited in section 4.2.2.

[BJ03] Reinaldo A. Bergamaschi and Yunjian W. Jiang. State-based power analysis for systems-on-chip. In *Proceedings of the 40th annual Design Automation Conference*, DAC '03, pages 638–641, New York, NY, USA, 2003. ACM. Cited in section 4.2.2.

[BKS08] Nicolas Blanc, Daniel Kroening, and Natasha Sharygina. Scoot: A tool for the analysis of systemc models. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 467–470. Springer, 2008. Cited in section 6.2.1.

[BLC02] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. Asm: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002. Cited in section 2.2.

[BLGJ91] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations: the signal language and its semantics. *Science of computer programming*, 16(2):103–149, 1991. Cited in section 3.2.3.

[BMF09] Tayeb Bouhadiba, Florence Maraninchi, and Giovanni Funchal. Formal and executable contracts for transaction-level modeling in systemc. In *Proceedings of the seventh ACM international conference on Embedded software*, pages 97–106. ACM, 2009. Cited in section 3.2.4.

[BMM13a] Tayeb Bouhadiba, Matthieu Moy, and Florence Maraninchi. System-level modeling of energy in TLM for early validation of power and thermal management. In *Design Automation and Test Europe (DATE)*, Grenoble, France, March 2013. 16.4% accepted as long-paper. Cited in sections 1.2.1, 4.2.4, and 4.2.4.

[BMM+13b] Tayeb Bouhadiba, Matthieu Moy, Florence Maraninchi, Jérôme Cornet, Laurent Maillet-Contoz, and Ilija Materic. Co-Simulation of Functional SystemC TLM Models with Power/Thermal Solvers. In *Virtual Prototyping of Parallel and Embedded Systems (VIPES)*, Boston, US, May 2013. Cited in sections 1.2.1 and 4.2.4.

[Boe05] Hans-J Boehm. Threads cannot be implemented as a library. In *ACM Sigplan Notices*, volume 40, pages 261–268. ACM, 2005. Cited in section 4.1.3.

[Bou07] Yussef Bouzouzou. Accélération des simulations de systèmes sur puce au niveau transactionnel. Diplôme de recherche technologique, UJF Grenoble, 2007. Cited in sections 4.1.4, 4.1.5, and 6.2.1.

[Bou10] Tayeb Bouhadiba. *42, Une Approche à Composants pour le Prototypage Virtuel des Systèmes Embarqués Hétérogènes*. These, Institut National Polytechnique de Grenoble - INPG, September 2010. Cited in section 3.2.4.

[BP09] Gérard Boudol and Gustavo Petri. Relaxed memory models: an operational approach. In *ACM SIGPLAN Notices*, volume 44, pages 392–403. ACM, 2009. Cited in section 4.1.3.

[BRS93] G. Berry, S. Ramesh, and R. K. Shyamasundar. Communicating reactive processes. In *In Proceedings of Twentieth ACM Symposium on Principles of Programming Languages*, pages 85–98. ACM Press, 1993. Cited in section 4.1.5.

[BSBM09] Ramzi Ben Salah, Marius Dorel Bozga, and Oded Maler. Compositional timing analysis. In *Proceedings of the seventh ACM international conference on Embedded software*, pages 39–48. ACM, 2009. Cited in section 5.5.

[BvL11] Harry Broeders and René van Leuken. Extracting behavior and dynamically generated hierarchy from SystemC models. In *Proceedings of the 48th Design Automation Conference*, DAC '11, pages 357–362, New York, NY, USA, 2011. ACM. Cited in section 3.1.2.

[CCF+05] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The astrée analyzer. *Programming Languages and Systems*, pages 140–140, 2005. Cited in section 2.1.

[CCG+02] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Computer Aided Verification*, pages 359–364. Springer, 2002. Cited in section 2.2.

[CCNR11] Daniele Campana, Alessandro Cimatti, Iman Narasamdya, and Marco Roveri. An analytic evaluation of systemc encodings in promela. In *Model Checking Software*, pages 90–107. Springer, 2011. Cited in section 3.2.1.

[CD13] Weiwei Chen and Rainer Dömer. Optimized out-of-order parallel discrete event simulation using predictions. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 3–8. EDA Consortium, 2013. Cited in section 4.1.5.

[CDS13] Inc. Carbon Design Systems. Carbon model studio, 2013. `http://www.carbondesignsystems.com/carbon-model-studio/`. Cited in section 1.2.1.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991. Cited in section 2.3.1.

[CGM+11] A. Cimatti, A. Griggio, A. Micheli, I. Narasamdya, and M. Roveri. KRATOS–a software model checker for SystemC. In *Computer Aided Verification*, pages 310–316. Springer, 2011. Cited in sections 3.1.3 and 3.2.1.

[CHD12] Weiwei Chen, Xu Han, and Rainer Dömer. Out-of-order parallel simulation for esl design. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 141–146. EDA Consortium, 2012. Cited in section 4.1.5.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004. Cited in sections 2.3.1 and 3.2.1.

[CM13] Chang-Chih Chen and Linda Milor. System-level modeling and microprocessor reliability analysis for backend wearout mechanisms. In *Design, Automation and Test in Europe (DATE)*, page 1615, 2013. Cited in section 4.2.1.

[CMCM+12] Jérôme Cornet, Laurent Maillet-Contoz, Ilija Materic, Sylvian Kaiser, Hela Boussetta, Tayeb Bouhadiba, Matthieu Moy, and Florence Maraninchi. Co-Simulation of a SystemC TLM Virtual Platform with a Power Simulator at the Architectural Level: Case of a Set-Top Box. In *Design Automation Conference*, page SESSION 10U: USER TRACK, San Francisco, US, June 2012. Cited in section 4.2.4.

[CMNR10] A. Cimatti, A. Micheli, I. Narasamdya, and M. Roveri. Verifying SystemC: a software model checking approach. In *Formal Methods in Computer-Aided Design (FMCAD), 2010*, pages 51–59. IEEE, 2010. Cited in section 3.2.1.

[Cor] Jonathan Corbet. Why the "volatile" type class should not be used. Technical Documentation of the Linux Kernel. Cited in section 4.1.3.

[Cor08] Jérôme Cornet. *Separation of Functional and Non-Functional Aspects in Transactional Level Models of Systems-on-Chip.* PhD thesis, Grenoble Institute of Technology, april 2008. Cited in sections 3.2.2 and 4.1.1.

[Des06] FZI Microelectronic System Design. Kascpar — Karlsruhe SystemC Parser suite, 2006. Cited in section 3.1.1.

[DGH+08] Markus Damm, Christoph Grimm, Jan Haase, Andreas Herrholz, and Wolfgang Nebel. Connecting systemc-ams models with osci tlm 2.0 models using temporal decoupling. In *FDL*, pages 25–30. IEEE, 2008. Cited in section 4.1.2.

[FGC+04] Görschwin Fey, Daniel Große, Tim Cassens, Christian Genz, Tim Warode, and Rolf Drechsler. Parsyc: An efficient SystemC parser. In *In Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI)*, pages 148–154, 2004. Cited in section 3.1.1.

[FM11a] Giovanni Funchal and Matthieu Moy. jTLM: an experimentation framework for the simulation of transaction-level models of systems-on-chip. In *Design, Automation and Test in Europe (DATE)*, 2011. acceptance rate : 314/950 = 33%. Cited in sections 1.2.1 and 4.1.4.

[FM11b] Giovanni Funchal and Matthieu Moy. Modeling of time in discrete-event simulation of systems-on-chip. In *ACM/IEEE Ninth International Conference on Formal Methods and Models for Codesign MEMOCODE*, Cambridge Royaume-Uni, 07 2011. Acceptance rate : 16/43 = 37%. Cited in sections 1.2.1 and 4.1.4.

[FMMCM11] Giovanni Funchal, Matthieu Moy, Laurent Maillet-Contoz, and Florence Maraninchi. Faithfulness considerations for virtual prototyping of systems-on-chip. In *3rd Workshop on: Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO)*, Heraklion, Crete, Greece, jan 2011. Cited in sections 1.2.1 and 4.1.3.

[Fun11] Giovanni Funchal. *Contributions to the Transaction-Level Modeling of Systems-on-a-Chip*. PhD thesis, Grenoble Institute of Technology, France, may 2011. Cited in section 4.1.3.

[GD03] Daniel Große and Rolf Drechsler. Formal verification of LTL formulas for SystemC designs. In *Circuits and Systems, 2003. ISCAS'03. Proceedings of the 2003 International Symposium on*, volume 5, pages V–245. IEEE, 2003. Cited in sections 3.2.1 and 3.2.1.

[GD05] Daniel Große and Rolf Drechsler. CheckSyC: An efficient property checker for RTL SystemC designs. In *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*, pages 4167–4170. IEEE, 2005. Cited in section 3.2.1.

[GDC92] Daniel D Gajski, Nikil D Dutt, and Allen CH. *High-level synthesis*, volume 34. Kluwer Boston, 1992. Cited in section 1.1.4.

[GH06] L. Gonnord and N. Halbwachs. Combining widening and acceleration in linear relation analysis. *Lecture Notes in Computer Science*, 4134:144, 2006. Cited in section 5.3.

[Ghe05] F. Ghenassia. *Transaction-level modeling with SystemC: TLM concepts and applications for embedded systems*. Springer Verlag, 2005. Cited in section 1.2.1.

[GHPS09] Hubert Garavel, Claude Helmstetter, Olivier Ponsini, and Wendelin Serwe. Verification of an Industrial SystemC/TLM Model using LOTOS and CADP. In *7th*

ACM-IEEE International Conference on Formal Methods and Models for Codesign MEMOCODE'2009, Cambridge, MA, US, 2009. Cited in section 3.2.1.

[GLD10] Daniel Große, Hoang M Le, and Rolf Drechsler. Proving transaction and system-level properties of untimed SystemC TLM designs. In *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on*, pages 113–122. IEEE, 2010. Cited in section 3.2.1.

[GLMS11] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. Cadp 2010: a toolbox for the construction and analysis of distributed processes. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 372–387. Springer, 2011. Cited in section 3.2.1.

[GR07] Denis Gopan and Thomas W. Reps. Guided static analysis. In *SAS*, volume 4634 of *Lecture Notes in Computer Science*, pages 349–365. Springer, 2007. Cited in section 2.3.2.

[Gre07] GreenSoCs. Greenbus, 2004-2007. Cited in section 1.2.1.

[GZ10] Sumit Gulwani and Florian Zuleger. The reachability-bound problem. In *PLDI*, pages 292–304. ACM, 2010. Cited in section 2.4.2.

[Har04] Reiner Hartenstein. The changing role of computer architecture education within cs curricula: invited presentation. In *Proceedings of the 2004 workshop on Computer architecture education: held in conjunction with the 31st International Symposium on Computer Architecture*, WCAE '04, New York, NY, USA, 2004. ACM. Cited in section 1.1.

[HCG+13] Claude Helmstetter, Jérôme Cornet, Bruno Galilée, Matthieu Moy, and Pascal VIVET. Fast and Accurate TLM Simulations using Temporal Decoupling for FIFO-based Communications. In *Design, Automation and Test in Europe (DATE)*, page 1185, Grenoble, France, Mar 2013. acceptance rate: 302/829 = 36.4% all categories. Cited in sections 1.2.1 and 4.1.2.

[Hel09] Claude Helmstetter. TLM.open: a SystemC/TLM Front-end for the CADP Verification Toolbox. Extended abstract for SBDCES workshop (http://unit.aist.go.jp/cvs/workshop/SBDCES.html) Work financed by the Multival project, October 2009. Cited in section 3.2.1.

[His01] Digh Hisamoto. Fd/dg-soi mosfet-a viable approach to overcoming the device scaling limit. In *Electron Devices Meeting, 2001. IEDM'01. Technical Digest. International*, pages 19–3. IEEE, 2001. Cited in section 1.1.3.

[HLR92] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying critical systems by means of the synchronous data-flow programming language lustre. *Transactions on Software Engineering*, pages 785–793, september 1992. Cited in section 1.1.4.

[HMG+06] Wei Huang, Student Member, Shougata Ghosh, Siva Velusamy, Karthik Sankaranarayanan, Kevin Skadron, Mircea R. Stan, Senior Member, and Senior Member. Hotspot: A compact thermal modeling method for CMOS VLSI systems. *IEEE Transactions on VLSI Systems*, 14:501–513, 2006. Cited in section 4.2.4.

[HMM12a] Julien Henry, David Monniaux, and Matthieu Moy. PAGAI: a path sensitive static analyzer. In Bertrand Jeannet, editor, *Tools for Automatic Program Analysis (TAPAS)*, 2012. Cited in sections 1.2.2, 2.4.2, and 2.4.3.

[HMM12b] Julien Henry, David Monniaux, and Matthieu Moy. Succinct representations for abstract interpretation. In *Static analysis Symposium (SAS)*, 2012. Acceptance rate: 40%. Cited in sections 1.2.2, 2.4, and 2.4.2.

[HMMCM06] Claude Helmstetter, Florence Maraninchi, Laurent Maillet-Contoz, and Matthieu Moy. Automatic generation of schedulings for improving the test coverage of systems-on-a-chip. *FMCAD*, pages 171–178, 2006. Acceptance rate: 21/90 = 23%. Cited in section 3.2.1.

[Hol91] Gerard J. Holzmann. *Design and validation of computer protocols*. Prentice-Hall, Englewood Cliffs, NJ, 1991. Cited in section 3.2.1.

[HT08] George Hagen and Cesare Tinelli. Scaling up the formal verification of Lustre programs with SMT-based techniques. In A. Cimatti and R. Jones, editors, *FMCAD*, pages 109–117, Portland, Oregon, 2008. IEEE. Cited in section 5.3.

[HYH+11] Chung-Yang (Ric) Huang, Yu-Fan Yin, Chih-Jen Hsu, Thomas B. Huang, and Ting-Mao Chang. SoC HW/SW verification and validation. In *Proceedings of the 16th Asia and South Pacific Design Automation Conference*, ASPDAC '11, pages 297–300, Piscataway, NJ, USA, 2011. IEEE Press. Cited in section 1.2.1.

[Jea03] B. Jeannet. Dynamic partitioning in linear relation analysis. application to the verification of reactive systems. *Formal Methods in System Design*, 2003. Cited in sections 1.1.4, 5.3, and 5.5.

[Jea09] Bertrand Jeannet. Relational interprocedural verification of concurrent programs. In *Software Engineering and Formal Methods, 2009 Seventh IEEE International Conference on*, pages 83–92. IEEE, 2009. Cited in section 3.2.2.

[JM09] B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *Computer Aided Verification*, pages 661–667. Springer, 2009. Cited in section 2.1.

[JPTY08] Bengt Jonsson, Simon Perathoner, Lothar Thiele, and Wang Yi. Cyclic dependencies in modular performance analysis. In *EMSOFT*, pages 179–188, New York, NY, USA, 2008. ACM. Cited in section 1.2.3.

[Kal12] Kalray. MPPA: Multi-Purpose Processor Array, 2012. Cited in section 1.1.1.

[KMS10] S. Kaiser, I. Materic, and R. Saade. ESL solutions for low power design. In *Proceedings of the International Conference on Computer-Aided Design*, pages 340–343. IEEE Press, 2010. Cited in section 4.2.2.

[KTBB06] Hamoudi Kalla, Jean-Pierre Talpin, David Berner, and Loïc Besnard. Automated translation of C/C++ models into a synchronous formalism. In *13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems (ECBS)*, pages 9–pp. IEEE, 2006. Cited in section 3.2.3.

[LA04] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004. Cited in section 2.3.1.

[LBT01] Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus*. Lecture Notes in Computer Science. Springer Verlag, 2001. Cited in sections 1.2.3 and 5.4.1.

[LGHD13] Hoang M. Le, Daniel Große, Vladimir Herdt, and Rolf Drechsler. Verifying SystemC using an intermediate verification language and symbolic simulation. In *DAC*, page 116. ACM, 2013. Cited in section 3.2.1.

[LPT09] Kai Lampka, Simon Perathoner, and Lothar Thiele. Analytic real-time analysis and timed automata: A hybrid method for analyzing embedded real-time systems. In *EMSOFT*. ACM, October 2009. Cited in sections 1.2.3, 5.1, and 5.2.

[LPT10] K. Lampka, S. Perathoner, and L. Thiele. Analytic real-time analysis and timed automata: a hybrid methodology for the performance analysis of embedded real-time systems. *Design Automation for Embedded Systems*, pages 1–35, June 2010. Cited in sections 1.2.3, 5.1, and 5.2.

[LPY97] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997. Cited in sections 1.2.3 and 5.2.

[LTDDS+11] Jean-François Le Tallec, Julien DeAntoni, Robert De Simone, Benoît Ferrero, Frédéric Mallet, Laurent Maillet-Contoz, et al. Combining systemc, ip-xact and uml/marte in model-based soc design. In *Workshop on Model Based Engineering for Embedded Systems Design (M-BED 2011)*, 2011. Cited in section 3.1.4.

[MB07] Florence Maraninchi and Tayeb Bouhadiba. 42: Programmable models of computation for a component-based approach to heterogeneous embedded systems. In *Proceedings of the 6th international conference on Generative programming and component engineering*, pages 53–62. ACM, 2007. Cited in section 3.2.4.

[MBF+12] Diego Melpignano, Luca Benini, Eric Flamand, Bruno Jego, Thierry Lepley, Germain Haugou, Fabien Clermidy, and Denis Dutoit. Platform 2012, a many-core computing accelerator for embedded socs: performance evaluation of visual analytics applications. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 1137–1142, New York, NY, USA, 2012. ACM. Cited in sections 1.1.1 and 4.2.1.

[McM93] Kenneth L McMillan. *Symbolic model checking*. Springer, 1993. Cited in section 2.2.

[MG11] David Monniaux and Laure Gonnord. Using bounded model checking to focus fixpoint iterations. In *Static analysis (SAS)*, volume 6887 of *lncs*, pages 369–385. springer, 2011. Cited in section 2.3.2.

[MM10a] Kevin Marquet and Matthieu Moy. PinaVM: a SystemC front-end based on an executable intermediate representation. In *International Conference on Embedded Software*, page 79, Scottsdale, USA, 10 2010. Acceptance rate: 29/89 = 32%. Cited in sections 1.2.1 and 3.1.4.

[MM10b] Kevin Marquet and Matthieu Moy. PinaVM: PinaVM Is Not A Virtual Machine. Published as Free Software (LGPL License), 2010. Cited in sections 1.2.1 and 3.1.4.

[MMC+08] Florence Maraninchi, Matthieu Moy, Jérôme Cornet, Laurent Maillet Contoz, Claude Helmstetter, and Claus Traulsen. SystemC/TLM Semantics for Heterogeneous System-on-Chip Validation. In IEEE, editor, *2008 Joint IEEE-NEWCAS and TAISA Conference 2008 Joint IEEE-NEWCAS and TAISA Conference*, Montréal Canada, June 2008. B.6.3, D.2.4, D.3.1, F.4.3, F.3.1, B.8.1. Cited in sections 1.2.2, 3.2.1, and 6.1.

[MMJ11] Kevin Marquet, Matthieu Moy, and Bertrand Jeannet. Efficient Encoding of SystemC/TLM in Promela. In *DATICS-IMECS*, Hong-Kong, 03 2011. Cited in sections 1.2.2, 3.2.1, 3.2.2, and 3.2.2.

[MMK10] Kevin Marquet, Matthieu Moy, and Bageshri Karkare. A theoretical and experimental review of SystemC front-ends. In *Forum for Design Languages (FDL)*, 2010. B.1.4, C.3 OpenTLM (Projet Minalogic). Cited in sections 1.2.1, 3.1.1, 3.1.4, and 6.1.

[MMMC05a] Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. LusSy: A toolbox for the analysis of systems-on-a-chip at the transactional level. In *International Conference on Application of Concurrency to System Design*, pages 26–35, June 2005. Acceptance rate: 23/45 = 51%. Cited in sections 1.2.2, 3.1.3, and 3.2.1.

[MMMC05b] Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. Pinapa: An extraction tool for SystemC descriptions of systems-on-a-chip. In *EMSOFT*, pages 317–324, September 2005. 25/88 = 28% accepted as regular papers. Cited in sections 1.2.1 and 3.1.3.

[MMMC06] Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. LusSy: an open tool for the analysis of systems-on-a-chip at the transaction level. *Design Automation for Embedded Systems*, 2006. special issue on SystemC-based systems. Cited in sections 1.2.1, 1.2.2, 3.1.3, and 3.2.1.

[MMZ+01] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001. Cited in section 3.2.1.

[Mok07] Leonid Mokrushin. Compositional analysis of timed systems by abstraction. PowerPoint Slides, 2007. Cited in sections 5.1 and 5.2.

[Moy05a] Matthieu Moy. Chapter 5.9, formal verification. In Frank Ghenassia, editor, *Transaction-Level Modeling with SystemC. TLM Concepts and Applications for Embedded Systems*, pages 190–206. Springer, 2005. Cited in sections 1.2.2 and 3.2.1.

[Moy05b] Matthieu Moy. *Techniques and Tools for the Verification of Systems-on-a-Chip at the Transaction Level*. PhD thesis, INPG, Grenoble, France, December 2005. Cited in sections 1.2.1, 1.2.2, 3.2.1, and 3.2.1.

[Moy11a] Matthieu Moy. Efficient and Playful Tools to Teach Unix to New Students. In *16th Annual Conference on Innovation and Technology in Computer Science Education ITiCSE*, Darmstadt Allemagne, 06 2011. Acceptance rate : 66/169 = 39%. Cited in section 6.1.

[Moy11b] Matthieu Moy. Unix-training: a set of tools to teach unix efficiently. Published as Free Software (LGPL License), 2011. `http://www-verimag.imag.fr/~moy/?Unix-training-a-set-of-tools-to`. Cited in section 6.1.

[Moy12a] Matthieu Moy. Modélisation transactionnelle des systèmes sur puces, 2012. Ensimag, Grenoble INP. Cited in section 1.2.1.

[Moy12b] Matthieu Moy. sc-during: tasks with duration for parallel programming in SystemC. Published as Free Software (LGPL License), 2012. Cited in sections 1.2.1 and 4.1.5.

[Moy13] Matthieu Moy. Parallel Programming with SystemC for Loosely Timed Models: A Non-Intrusive Approach. In *The Design, Automation, and Test in Europe (DATE)*, Grenoble, France, March 2013. 16.4% accepted as long-paper. Cited in sections 1.2.1 and 4.1.5.

[MPA11] Ons Mbarek, Alain Pegatoquet, and Michel Auguin. A methodology for power-aware transaction-level models of systems-on-chip using upf standard concepts. In *Integrated Circuit and System Design. Power and Timing Modeling, Optimization, and Simulation*, pages 226–236. Springer, 2011. Cited in section 4.2.2.

[MS07] Pierre Michaud and Yiannakis Sazeides. ATMI: analytical model of temperature in microprocessors. *Third Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)*, 2007. Cited in section 4.2.3.

[Ope08] Open SystemC Initiative (OSCI). *OSCI TLM-2.0 Language Reference Manual*, June 2008. Cited in section 1.2.1.

[Ope11] Open SystemC Initiative. *IEEE 1666 Standard: SystemC Language Reference Manual*, 2011. Cited in sections 1.2.1 and 4.1.2.

[PCT08] Linh TX Phan, Samarjit Chakraborty, and PS Thiagarajan. A multi-mode real-time calculus. In *Real-Time Systems Symposium, 2008*, pages 59–69. IEEE, 2008. Cited in section 5.1.

[PCTT07] Linh T.X. Phan, Samarjit Chakraborty, P.S. Thiagarajan, and Lothar Thiele. Composing functional and state-based performance models for analyzing heterogeneous real-time systems. In *RTSS*, pages 343–352, Los Alamitos, CA, USA, 2007. IEEE Computer Society. Cited in sections 1.2.3 and 5.1.

[PHZ12] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 29–42, New York, NY, USA, 2012. ACM. Cited in section 4.2.1.

[PLT11] Simon Perathoner, Kai Lampka, and Lothar Thiele. Composing heterogeneous components for system-wide performance analysis. In *DATE*, pages 842–847. IEEE, 2011. Cited in sections 5.1 and 5.2.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977. Cited in section 5.3.

[PS08] Olivier Ponsini and Wendelin Serwe. A Schedulerless Semantics of TLM Models Written in SystemC via Translation into LOTOS. In *Formal Methods*, Turku, Finlande, 2008. Cited in section 3.2.1.

[SAK07] M. Slee, A. Agarwal, and M. Kwiatkowski. Thrift: Scalable cross-language services implementation. *Facebook White Paper*, 2007. Cited in sections 4.2.4 and 4.2.4.

[SLPH10] Christoph Schumacher, Rainer Leupers, Dietmar Petras, and Andreas Hoffmann. parSC: synchronous parallel SystemC simulation on multi-core host architectures. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 241–246. ACM, 2010. Cited in section 4.1.5.

[SMV09] Scott S Sirowy, Bailey Miller, and Frank Vahid. Portable systemc-on-a-chip. In *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 21–30. ACM, 2009. Cited in section 3.1.3.

[SN07] Thorsten Schubert and Wolfgang Nebel. The quiny systemctm front end: Self-synthesising designs. In *Advances in Design and Specification Languages for Embedded Systems*, pages 93–109. Springer, 2007. Cited in section 3.1.2.

[Sny12] Wilson Snyder. Verilator, 2012. Cited in section 1.2.1.

[ST05] STMicroelectronics SPG Team. Tac: Transaction accurate communication channel, 2005. Cited in section 1.2.1.

[SWD13] Jannis Stoppe, Robert Wille, and Rolf Drechsler. Data extraction from SystemC designs using debug symbols and the SystemC API. In *IEEE Computer Society Annual Symposium on VLSI*. IEEE, 2013. Cited in section 3.1.2.

[TCMM07] Claus Traulsen, Jérôme Cornet, Matthieu Moy, and Florence Maraninchi. A SystemC/TLM semantics in Promela and its possible applications. In *14th Workshop on Model Checking Software SPIN*, July 2007. Cited in sections 1.2.2, 3.2.1, 3.2.2, and 3.2.2.

[TCN00] Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. Real-time calculus for scheduling hard real-time systems. In *International Symposium on Circuits and Systems ISCAS 2000*, volume 4, pages 101–104, Geneva, Switzerland, march 2000. Cited in section 1.2.3.

[Upp07] Uppsala University. Cats tool, 2007. Cited in section 1.2.3.

7.2 Students

Following is the list of students that I supervised or co-supervised. More details about each student (report, slides, ...) can be found online at `http://www-verimag.imag.fr/~moy/?-Students-`.

Giovanni Funchal (June 2007): Research master, 2nd year, co-supervised with Florence Maraninchi. *Comparaison de traces de simulations à différents niveaux d'abstractions.*

Giovanni Funchal (September 2007 - November 2011): CIFRE Ph.D, Verimag/STMicroelectronics, co-supervised with Florence Maraninchi (≈ 70% by myself). *Contributions à la*

Modélisation Transactionnelle des Systèmes-sur-Puce.

Nabila Abdessaied (January 2009 - July 2009): Research master, 2nd year, co-supervised with Giovanni Funchal. *développement d'un simulateur alternatif à SystemC écrit en Java.*

Julien Henry (December 2010 - June 2011): Research master, 2nd year, co-supervised with David Monniaux. *Static Analysis by Path Focusing.*

Samuel Jones (December 2010 - June 2011): Research master, 2nd year, co-supervised with Claire Maiza. *Optimistic Parallelisation of SystemC.*

Julien Henry (August 2011 -): Ph.D, co-supervised with David Monniaux ($\approx$ 50% by myself). *Analyse statique par interprétation abstraite et procédures de décision.*

Loïc Crétin (February - May 2008): Master 1, Ensimag, co-supervised with David Monniaux. *Développement d'un mini analyseur statique de code intégré dans Eclipse.*

Madhav Jha (May - July 2007): Bachelor of Engineering intern. *Connection de la chaine d'outils LusSy au model-checker NuSMV.*

Yanhong Liu (February 2008 - July 2009): Post-doc on the FoToVP project, co-supervised with Karine Altisen. *Interface entre le « Real-Time Calculus » et les automates temporisés.*

Kevin Marquet (September 2008 - February 2010): Post-doc on the Minalogic OpenTLM project, co-supervised with Florence Maraninchi. *Vérification formelle de programmes écrits en SystemC au niveau TLM.*

Bageshri Karkare (May 2009 - September 2009): Post-doc on the Minalogic OpenTLM project. *Utilisation de la forme SSA dans un front-end SystemC.*

Romain Salles (February - May 2009): Master 1, Ensimag, co-supervised with David Monniaux. *Model-checking de programmes Java.*

Julien Henry (February - May 2010): Master 1, Ensimag, co-supervised with David Monniaux. *Analyse de programmes par SMT-solving.*

Xavier Jean (April 2009 - July 2009): École Polytechnique intern, 2nd year, co-supervised with Karine Altisen. *Étude des performances dans les systèmes embarqués, entre simulation numérique et solution équationnelle.*

Rafael Velasquez (February - June 2010): Master 1, UJF, co-supervised with Giovanni Funchal. *Définition de portabilité en termes de modèle d'exécution pour la simulation des systèmes sur puces.*

Mohamed El Aissaoui (February - May 2010): Master 1, Ensimag, co-supervised with Giovanni Funchal. *Modélisation du temps dans un simulateur pour systèmes sur puces.*

Pierre-Yves Delahaye (February - May 2010): Master 1, Ensimag, co-supervised with Kevin Marquet. *Transformation de programmes SystemC vers langage de vérification.*

Ranjan Ravi (May - July 2010): Bachelor of Engineering intern, co-supervised with Kevin Marquet. *Amélioration du back-end SPIN de PinaVM.*

Marc Pegon (February - May 2011): Master 1, Ensimag, co-supervised with David Monniaux. *Analyse de programmes par interprétation abstraite.*

Mohamed Zaim-Wadghiri (February - May 2011): Master 1, Ensimag, co-supervised with Claire Maiza. *Techniques de compilation dédiées pour le simulateur SystemC.*

Si-Mohamed Lamraoui (December 2010 - June 2011): TER and "Magistère" UJF, co-supervised with Claire Maiza. *Techniques de compilation dédiées pour SystemC en utilisant l'infrastructure de compilation LLVM.*

Guillaume Sarrazin (February - May 2011): Master 1, Ensimag, co-supervised with Karine Altisen. *Étude d'un algorithme de fermeture causale sur des courbes d'arrivée ayant des parties affines.*

Henry-Joseph Audeoud (June - July 2011): Excellence internship, bachelor 1st year, UJF, co-supervised with Claire Maiza. *Comparaison d'approches de parallélisation de SystemC.*

Valentin Bousson (July - August 2010): Ensimag 1st year intern, co-supervised with Claire

Maiza. *Dévelopement d'application en langage synchrone pour un robot LeGO NXT.*
Tayeb Bouhadiba (June 2011 - September 2012): Post-doc, HeLP project, co-supervised with Florence Maraninchi. *Modélisation de la consommation d'énergie au niveau TLM.*
Eduardo León (February - May 2013): Master 1, Ensimag, co-supervised with Claude Helmstetter. *Visualisation graphique de traces de simulation de systèmes sur puces.* o
Francesco Bongiovanni (May - June 2013): Post-doc, HeLP project. *Portage de sc-during en C++2011.*
Xavier Poczekajlo (February - May 2013): Master 1, Ensimag. *Optimisations de performances de simulateurs sur machines multi-cœurs.*
Claude Helmstetter (October 2012 - June 2013): Post-doc, HeLP project, co-supervised with Florence Maraninchi. *Modélisation de la consommation d'énergie au niveau TLM.*
Guillaume Sergent (February 2014 - June 2014): Research master, 2nd year. *Dedicated Compilation Techniques for SystemC.*

Index

Abstract

Modern embedded systems have reached a level of complexity such that it is no longer possible to wait for the first physical prototypes to validate choices on the integration of hardware and software components. It is necessary to use models, early in the design flow. The work presented in this document contribute to the state of the art in several domains.

First, we present some verification techniques based on abstract interpretation and SMT-solving for programs written in general-purpose languages like C, C++ or Java.

Then, we use verification tools on models written in SystemC at the transaction level (TLM). Several approaches are presented, most of them using compilation techniques specific to SystemC to turn the models into a format usable by existing tools.

The second part of the document deal with non-functional properties of models: timing performances, power consumption and temperature. In the context of TLM, we show how functional models can be enriched with non-functional information.

Finally, we present contributions to the modular performance analysis (MPA) with real-time calculus (RTC) framework. We describe several ways to connect RTC to more expressive formalisms like timed automata and the synchronous language Lustre. These connections raise the problem of causality, which is defined formally and solved with the new causality closure algorithm.

Keywords: Modeling, parallelism, System-on-a-Chip, embedded systems, formal verification, model-checking, abstract interpretation, SMT-solving, compilation, SystemC, TLM, Lustre, real-time calculus, causality .

Résumé

Les systèmes embarqués modernes ont atteint un niveau de complexité qui fait qu'il n'est plus possible d'attendre les premiers prototypes physiques pour valider les décisions sur l'intégration des composants matériels et logiciels. Il est donc nécessaire d'utiliser des modèles, tôt dans le flot de conception. Les travaux présentés dans ce document contribuent à l'état de l'art dans plusieurs domaines.

Nous présentons dans un premier temps de nouvelles techniques de vérification de programmes écrits dans des langages généralistes comme C, C++ ou Java.

Dans un second temps, nous utilisons des outils de vérification formelle sur des modèles écrits en SystemC au niveau transaction (TLM). Plusieurs approches sont présentées, la plupart d'entre elles utilisent des techniques de compilations spécifiques à SystemC pour transformer le programme SystemC en un format utilisable par les outils.

La seconde partie du document s'intéresse aux propriétés non-fonctionnelles des modèles : performances temporelles, consommation électrique et température. Dans le contexte de la modélisation TLM, nous proposons plusieurs techniques pour enrichir des modèles fonctionnels avec des informations non-fonctionnelles.

Enfin, nous présentons les contributions faites à l'analyse de performance modulaire (MPA) avec le calcul temps-réel (RTC). Nous proposons plusieurs connections entre ces modèles analytiques et des formalismes plus expressifs comme les automates temporisés et le langage de programmation Lustre. Ces connexion posent le problème théorique de la causalité, qui est formellement défini et résolu avec un algorithme nouveau dit de « fermeture causale ».

Mots Clés : Modélisation, parallélisme, Systèmes sur puce, systèmes embarqués, vérification formelle, model-checking, interprétation abstaite, SMT, compilation, SystemC, TLM, Lustre, calcul temps-réel, causalité .