Anca Molnos · Christian Fabre *Editors*

Model-Implementation Fidelity in Cyber Physical System Design

Springer

Model-Implementation Fidelity in Cyber Physical System Design

Anca Molnos • Christian Fabre
Editors

Model-Implementation Fidelity in Cyber Physical System Design

Springer

Editors
Anca Molnos
Campus MINATEC
CEA, Grenoble, France

Christian Fabre
Campus MINATEC
CEA, Grenoble, France

ISBN 978-3-319-47306-2 ISBN 978-3-319-47307-9 (eBook)
DOI 10.1007/978-3-319-47307-9

Library of Congress Control Number: 2016958303

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

The trend towards large-scale deployments of cyber-physical systems (CPSs) makes analysis of computing and physical world interactions of paramount importance. During the development of the software, designers model the physical world that the future CPS will interact with. Some of these models represent the external physical world that the CPS will monitor and control, and others represent a physical part of the CPS itself: its computing platform, sensors and actuators.

Models are built with various techniques and granularity. Several of these models are used at various stages of the CPS lifecycle, from the early stages of specification to development and all the way to production time when a CPS is actually deployed. Furthermore, they are involved not only in the development of the software but also in the assessment of functional and non-functional properties of the full CPS.

As a consequence, CPS software development raises several questions on the *fidelity of models*:

- What is the fidelity of a model with regard to the part of the physical world it represents?
- To what extent is a model's fidelity adequate, i.e. accurate enough but not too complex or costly, with regard to the way the model will be used by the CPS software?
- What is the impact of a model's fidelity on key properties of the full CPS, like dependability, performance or energy consumption?

These three questions lay beneath the authors' contributions. The chapters address several crucial CPS design aspects such as cross-application interference, parsimonious modelling and trustful code production. The book describes a wide range of solutions from simulation for extra-functional properties, extension of programming techniques, model-driven development (MDD), resource-driven modelling and quantitative and qualitative verification based on statistics and formal proofs. These solutions are applied to several CPS design techniques: mixed criticality, communication protocols and computing platform simulation. Tentative answers are presented from very different communities, such as compiler construction, power/temperature modelling of digital devices, high-level performance analysis, code/device certification, etc.

The target audience is researchers and engineers in the field of CPS development and validation. They will have the opportunity to learn what is the common practice in these fields and, more importantly, to make the links between them. Trends and open research issues presented in this book can be an inspiration for future research.

Grenoble, France
August 29, 2016

Anca Molnos
Christian Fabre

Introduction

This book presents different modelling and analysis techniques used for CPS development, such as model-driven development (MDD), resource-driven development, statistical analysis and proofs of simulator implementation. Each chapter describes the modelling techniques used and the analysis enabled thereby and discusses *model fidelity* in this specific context.

The first chapter overviews some of the sought challenges for building faithful embedded systems models, especially the growing demand for using formal models for dealing with, e.g., performance. The impact of the hardware part of the system on performance is discussed, and a probabilistic interpretation is proposed to build appropriately abstract models towards trustworthy analysis. Such a view is useful to investigate the system performance as it provides a formal and parsimonious framework.

The second chapter offers a view on how to manage the fidelity of a model's representation in order to control its complexity. This approach includes two concrete and related methods targeting two aspects of the problem. Dynamic resource graphs highlight the dependencies between system resources and describe a system's progression as resource and dependency evolution steps. This forms a theoretical foundation for tracking the parameters that can be regarded as resources, e.g. power consumption, time and computation units. With this resource-oriented view of a system, a hierarchical modelling method emphasising cross-layer cuts is established. This method facilitates parameter-proportional modelling to achieve fidelity vs. complexity trade-offs in models. Use-case simulation and state space analysis help to validate the approach.

The third chapter covers the power, thermal and reliability problems brought by the extreme device, core and multicore scaling that we face today. A system engineer should be aware of any possible cross-application interferences with respect to timing, power and thermal properties as soon as possible in the design process. Power and temperature management should be dealt with without introducing unwanted interference. For this reason, the extra-functional properties need to be modelled and analysed at the system level, because they can strongly affect the overall quality of service (QoS)—performance and battery lifetime—or even cause the system to fail meeting its real-time and safety requirements. This covers the specification of platform properties (extra-functional model) as well as the dynamic capturing, processing and extraction of power/temperature information during the simulation. Especially closing the loop back to the application and run-time services is an important feature for complex heterogeneous hardware platforms and software stacks. As an example, a battery-powered mixed-critical avionics system, running a safety-critical flight control application and a performance critical image processing application on the same multicore system-on-chip (SoC) is presented.

Chapter 4 addresses the performance analysis in relation with the programming models for the embedded domain. In particular, the focus is on synchronous data flow (SDF) graphs that are often the computational model of choice for specification, analysis and automated synthesis of parallel streaming kernels targeting embedded multiprocessor system-on-chip (MPSoC) platforms. The chapter discusses several limitations of the SDF graphs in the context of conventional parallel software synthesis methodologies and highlights the associated degradation of fidelity, in terms of analysis' accuracy and synthesised software performance. Subsequently, the chapter proposes several extensions to the strict notion of SDF graph model that address the identified issues. An extensive empirical evaluation, which underscores the model limitations and the effectiveness of the approach, is included.

Chapter 5 presents a simulation view on the process of verification of MPSoC. The virtual prototyping framework, SimSoC, is proposed. SimSoC is a full system simulation framework based on SystemC and transaction-level modelling (TLM). It takes as input a binary executable file, which can be a full operating system, and simulates the behaviour of the target hardware on the host system. It is using internally dynamic binary translation from target code to host code to simulate the application software. A potential issue with simulators is that they might not accurately simulate the real hardware. This gap is filled by proving that an instruction set simulator coded in C (of an ARM processor, in our case) is a high-fidelity implementation of the processor architecture. The first part of the chapter presents the general architecture and features of SimSoC. The second part describes the proof of the ARM simulator.

Chapter 6 addresses the design and verification of mixed-critical systems from models of computation to describe an application all the way to the implementation on a hardware platform. The chapter describes a compositional method to design applications independently and then to execute them without interference. It defines a formal modelling framework as a suitable entry point for application design. The models are executable, which enables early detection of specification errors, and include the formal properties of the applications based on well-defined models of computation. This is combined with a predictable MPSoC platform template that has a supporting design flow. The structure and behaviour of the application models are exported to an intermediate format via introspection which is iteratively transformed for the backend flow. The problems arising in this transformation are identified and appropriate solutions are provided. The design flow is demonstrated by a system consisting of two streaming applications.

Chapter 7 tackles the problem of qualitative and quantitative verification of an embedded system via a holistic development approach. This includes visual modelling of the system and its environment, qualitative and quantitative verification of the model and automated executable code generation. An application with electronic tags is utilised to illustrate the approach. To this end, pState, a tool for the design and verification of hierarchical state machines, is extended with probabilistic transitions, costs/rewards and state invariants, called pCharts. From a pChart, pState generates input code for a probabilistic model checker in the form of either a Markov decision process (MDP) or a probabilistic timed automaton (PTA). On the generated model, qualitative and quantitative properties can be verified to help assess its fidelity. From sub-charts without probabilistic transitions, pState can generate executable code in C or assembly language.

Finally, Chap. 8 focusses on reducing the design gap between design complexity and design productivity associated with the drastic increase of functionality, implemented as software or dedicated hardware, in embedded MPSoCs. The trend is to increase the level of abstraction at which designers and CAD tools work. To deal with this problem, starting the design process from high-level UML models combined with functional code using (i.e. in C/C++) the different system components is presented. Video processing is one of the areas where high-level modelling and analysis based on UML may have a wider impact. In this chapter, MDD using UML/MARTE is proposed to support the specification and analysis of a positioning system for “recreated reality” applications.

The chapters describe techniques and goals characteristic for several engineering contexts: modelling of computing and communication, proof architecture models and statistically based validation techniques.

Contents

Chapter 1
Building Faithful Embedded Systems Models: Challenges and Opportunities

Ayoub Nouri, Marius Bozga, and Saddek Bensalem

1.1 Introduction

Embedded systems (ES) have deeply impacted our daily lives and contributed to draw a completely new lifestyle where mobility, rapidity, and connectivity are the keywords. The substantial advances in the integrated circuits and the networks bandwidth have contributed to democratize these systems which became ubiquitous. According to the Artemis Strategic Research Agenda 2011,[1] it is estimated that there will be over 40 billion devices worldwide, that is 5–10 embedded devices per person on earth by 2020. While a large portion of ES is dedicated for customer electronics (entertainment and personal use), they are becoming also essential for companies and even for governments and states as they represent an important leverage for innovation and competitiveness. Domains benefiting from ES assets, such as transportation, national security, health care, and education, to mention but a few, are wide and steadily increasing.

The great and various benefits of ES come at the price of an increasing complexity to design them. More burden is thus put on designers that have

[1] ftp.cordis.europa.eu/pub/technology-platforms/docs/sra-2011-book-page-by-page-9.pdf

A. Nouri (✉)
Université Grenoble Alpes, F-38000 Grenoble, France

CEA, LETI, MINATEC Campus, F-38054 Grenoble, France
e-mail: ayoub.nouri@imag.fr

M. Bozga • S. Bensalem
VERIMAG, Université Grenoble Alpes, F-38000 Grenoble, France

VERIMAG, CNRS, F-38000 Grenoble, France
e-mail: marius.bozga@imag.fr; saddek.bensalem@imag.fr

A. Molnos, C. Fabre (eds.), *Model-Implementation Fidelity in Cyber Physical System Design*, DOI 10.1007/978-3-319-47307-9_1

to produce systems in less time with ever reducing costs. Whereas designing mixed hardware/software systems that provide sophisticated services is inherently challenging, additional constraints such as the shrinking time to market and costs optimization make it even harder. Furthermore, ES are increasingly used in safety-critical setups involving human lives and wealth. Thus, formally ensuring their functional correctness is becoming primordial.

As our reliance on these systems increases, so does the expectation that they will provide us with more mobility and ensure high-performance response. The major breakthrough of ES in our everyday lives has resulted on a shift from task-specific settings, e.g., control of an industrial robot, to more programmable configurations running various functionalities, targeting mass consumption, and often battery supplied. The former rely on dedicated hardware ensuring low power consumption, low cost, real-time constraints, and silicon efficiency, whereas the latter require programmable circuits with an increasing computation power, which are less efficient, consume more energy, and are harder to program.

The design of these systems generally falls within one of three configurations[2]: (1) *hardware-centric*: which aims at finding the most appropriate hardware architecture for a specific domain of application, e.g., multimedia, (2) *software-centric*: which tackles the issue of designing and deploying functionally correct and efficient applications on a given hardware architecture, and (3) *co-design*: which starts from abstract functional specifications and tries to figure out the best partition of these functionalities into software and hardware components.

In this chapter, we focus on the design of embedded software applications for a given hardware architecture [30]. Whereas several advances have been performed in this context [23, 27, 28, 34, 50, 58], many open challenges are still ahead, especially with the advent of multi- and many-cores architectures, e.g., how to distribute and map software applications onto these platforms for an optimal utilization and to satisfy performance requirements, e.g., energy?

This chapter considers the design of embedded software following a model-based approach and using formal techniques. In model-based design, the whole process is driven by models and all the choices are made upon them. Hence, they must satisfy two important conditions for a successful design. First, they must be faithful, that is to capture the real behavior of the system, in term of functionality and performance. Second, the built models must have a clear interpretation, i.e., they must have a formally defined semantics, to enable unambiguous/trustworthy analysis and then enable to make well-founded choices. Relying on models that do not respect these conditions leads to inconsistent decisions and later to poor designs.

In this work, our concern are the earliest phases of the design as they have the greatest impact on the rest of the process. We precisely focus on modeling performance aspects, which are inherently challenging and hard to faithfully predict during the early stages of ES design. Unlike classical programs design where performance is evaluated using complexity theory with respect to abstract machine,

[2]These settings may coexist within the same design process, e.g., at different phases.

i.e., Turing machine, the performance of ES is induced by the combination of the software and hardware parts of the system, e.g., the execution time of a Fourier transform on a processing unit, the communication delay of a bus or a Network on Chip (NoC), the amount of consumed energy induced by that function on the corresponding hardware.

Building faithful, formal, and high-level system models that capture the functional and the performance aspects entails several challenges, e.g., how to access (predict) the real performance evolution of the system during the early design phases? Assuming that such information is available, what is the appropriate model to characterizes it faithfully, and how? Additional challenges concern the appropriate level of abstraction of the performance model. In addition of being determinant for applying formal analysis techniques, the choice of the adequate level of abstraction is essential to enable understanding and mastering the complexity of the system under design, especially early in the process.

An important characteristic that must be considered when dealing with performance of ES is variability. ES performance is ideally constant, albeit in practice, it shows fluctuation, which should be taken into consideration for a trustworthy analysis. The environment where the ES is deployed has an important impact on performance and it can be seen twofold. First, is the input data to the system, e.g., different video qualities for a multimedia player. Variable inputs generally provokes a variable performance, although some systems expose a constant behavior when varying input data (data-independent systems). The second component within the environment includes all the external factors which may affect its function and/or performance, e.g., unusual external temperature. Another important source of fluctuation, which will be our focus in this chapter, is the hardware (and networked) part of the system, e.g., buses and memory.

Ideally, the variability induced by the above factors is deterministically characterized, which implies to build detailed models of the environment and the hardware architecture. However, besides being in contradiction with the objective of building high-level models, this is generally unfeasible. Precisely modeling the environment (including inputs) is unfeasible unless the system is designed for specific tasks and is targeted to a well-known and controlled environment. Similarly, building formal detailed models of the underlying hardware is challenging. First, it requires to have detailed specifications, which are rarely available during early design phases. Assuming that such details are available, this will induce an important understanding/interpretation effort and consequently a considerable time and produces huge models. This eventually leads to an ad hoc approach which is tedious and error prone. Abstraction of details is thus a must. That is, we need models and methods that enable characterizing performance[3] and to use it for building formal high-level system models (including the functional behavior) towards trustworthy analysis.

[3]Our view of characterizing performance is to capture the gist of performance evolution formally and in a parsimonious way.

The vision supported in this work is to interpret the performance evolution probabilistically. We argue that such a view provides a natural and faithful abstraction in addition of being the unique mathematically sound framework for capturing variability and its uncertainty. Probabilistic modeling is a natural choice to abstract details either because we don't care about them for the moment, e.g., early design stages, or because we are not able to handle all of them, e.g., no access, too complex, or it takes too much time.

The chapter aims at providing, without claiming to be exhaustive, an overview of the sought challenges and promising techniques for building faithful, formal, high-level probabilistic software/hardware models for performance evaluation. This work is motivated by a steadily growing research results providing efficient formal probabilistic techniques that can be used for performance evaluation [4, 26, 63]. We consider that among the obstacle for using such techniques are the absence of systematic techniques to obtain faithful formal models. We detail in Sect. 1.2 the issues related to faithfully characterizing ES performance. Then we review in Sect. 1.3 some existing methods and models for gathering and characterizing performance. In Sect. 1.4, we overview a recent work that provides an attempt to answer the underlying challenges following a probabilistic view. We discuss the assumptions and restrictions made in this work and provide in Sect. 1.5 future directions and sought opportunities to overcome them.

1.2 Challenges for ES Performance

Performance of ES can be thought in term of several aspects, e.g., timing behavior (computation and communication), energy consumption, memory utilization, or global throughput. These aspects are necessary together with functional behavior to enable trustworthy system analysis. In this section we highlight some factors that may affect ES performance. We precisely focus on the impact of the hardware part of the system.[4]

Hardware architectures complexity is steadily increasing due to the massive integration of sophisticated functionalities, e.g., dynamic energy management. Due to the transistors integration limit and the thermal wall, a shift towards multi- and many-cores architectures has been operated. These provide a considerable computation power; however, they raise several problems for efficiently designing and implementing software on them. More importantly, they pose together with other hardware mechanisms such as caches, several challenges for the early estimation of systems performance. To illustrate these challenges, we discuss hereafter the impact of various hardware mechanisms on the system performance. We mainly focus on timing and energy aspects.

[4]Challenges related to modeling the environment are out of the scope of this chapter.

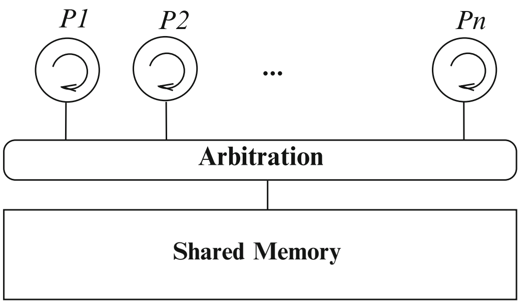


Fig. 1.1 Memory contention and arbitration

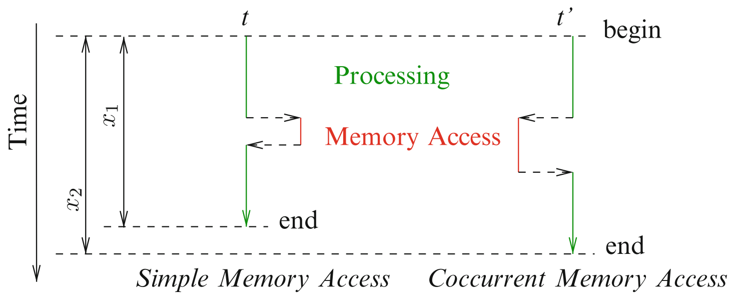


Fig. 1.2 Contention impact on execution time

1.2.1 Memory Contention

Consider the problem of characterizing the execution time or the consumed energy of a process P (executing some computation function, i.e., a set of sequential instructions) running on a specific processing unit. The latter is first assumed to exclusively execute P, i.e., no scheduling algorithm is required on top of it, and thus no context swapping overhead is induced for the moment. Data to be processed by P is stored in a shared memory which is accessible by different processing units executing similar tasks as shown in Fig. 1.1. Thus, the execution time of P will vary depending on the number of processes and the way they are accessing the shared memory each time. Formally, if at time t, the processing unit executing P tries to access the memory alone, its execution time will be x_1, which is the time of executing the instructions of P when directly accessing the memory and getting the required data. Now, assume that at time $t' \neq t$, the processing unit executing P tries to access the memory simultaneously with $n - 1$ others concurrent processing units (in competition). The execution time of P will be $x_2 = x_1 + \epsilon$, where ϵ is a time overhead encompassing the arbitration time and the waiting time (stall) to access the memory as illustrated in Fig. 1.2. The overhead ϵ varies with respect to the number of concurrent processes simultaneously accessing the memory and the implemented arbitration policy. Hence, different executions of this setting provide different execution times of P.

1.2.2 Memory Architectures

In the previous paragraph, we considered a relatively simple setting where the shared memory is designed as a single addressed space accessible by all the processing units in a similar way. With the progress of the hardware design, more involved memory architectures were proposed to overcome the memory access latency. It is known that the processing units are generally faster than the memory and thus their access represents a bottleneck for computation. Architectures such as the Tightly Coupled Memory (TCM) or the Non-Uniform Memory Access (NUMA) aim at reducing the access latency by providing parallel access to different memory banks (different addressed spaces). The main idea behind these architectures is to divide the shared memory into different portions accessible in parallel (a portion for each processing unit), with a fast local access and slower distant one. The point is that such solution may reduce latency by exposing more parallelism for access. However, this should be carefully mitigated since the concurrency effect is still important as it depends on how the required data (by each process) is mapped onto the portions of the memory. Concretely, when the data required by two or more processes is mapped onto the same portion, a concurrent access will be necessary to fetch it. Thus, arbitration is still needed at this level. In these examples, note that it is increasingly hard to follow and understand the impact on the time and energy evolution of the system, although, we only considered data memory. Similar technologies are generally used to store instructions. Furthermore, other architectures propose structured memories with several levels and hence increasing the complexity of understanding and estimating the impact on the system performance.

1.2.3 Caches and DMAs

In the setting shown in Fig. 1.1, assume that we introduce caches and/or DMAs. These hardware components will further impact the variation of the execution/-communication times and similarly the energy of the system. For instance, for caches, depending on miss or hit situations, a processing unit will require to access the main (shared or local) memory or to get the data from the cache which is in turn a shared resource and requires arbitration when accessed concurrently. As for the DMA, its impact on the system performance can be interpreted as follow. The DMA enables the processing elements to perform data transfer in parallel to computation. Concretely, the processing unit only initiates the communication and another specialized hardware component will perform it while the processing unit continues its computation. When the transfer is done, mechanisms such as hardware interruptions are used to notify the processing element. This has the advantage to reduce communication overhead in the processing element, however, it consumes more energy as it uses specialized hardware. Moreover, it makes the task of observing the exact communication and computation times more involved.

1.2.4 Multi-processing, Pipelining, and Others

In addition to the previous hardware functionalities, a processing element is generally able to execute several tasks in (pseudo-)parallel. This requires to use a scheduling policy to manage them (or synchronization/mutual exclusion mechanisms). A context swap between the executing processes (save/restore of stack) is thus operated, which generates an additional time and energy overhead. More complicated settings are induced by more sophisticated mechanism such as pipelines (with different levels), execution speculation mechanisms, and dynamic frequency scaling procedures. Techniques such as tasks migration, which moves processes from a processing element to another or onto a completely different cluster to equilibrate energy consumption or temperature over a chip/cluster, add a huge complexity to the system.

Modeling the previously discussed hardware aspects (and many others which are not discussed here) in details is not possible neither recommended during the early stages of ES design. High-level models have to be abstract, easy to understand, and rapid to build and to analyze. As shown in the previous paragraphs, hardware aspect might be too complex to understand and to model in detail, especially for application development teams. These have as mission to design and implement applications which satisfy specific performance requirements in addition of being functionally correct. This should not be conditioned by a detailed comprehension and definitely not a detailed modeling of the functional and the non-functional behavior of target platforms. Nevertheless, performance aspects should be somehow taken into account early in the design to enable faithful modeling and later-on a trustworthy analysis towards good designs.

In the next section, we review some state-of-the-art techniques for modeling performance of ES. We discuss their advantages and weaknesses and argue in favor of probabilistic techniques that we believe to be well-suited for characterizing ES performance faithfully.

1.3 Performance Modeling: State of the Art

In this section, we review methods and models used in the literature for ES performance characterization. The section will be organized around three important issues, namely (1) the used methods for gathering performance details during early design stages, (2) given raw performance data, the usually utilized models to model performance, and finally, (3) the methods used to build such models.

1.3.1 Techniques for Gathering ES Performance

The state-of-the-art techniques for gathering low-level performance information in early design phases can be classified into three different families. The first uses human expertise or documentation such as constructors data sheet of hardware architectures [23]. When available, this provides an easy way to get performance information, albeit it does not necessarily include details about the system performance, e.g., caches impact. The second used technique is based on source/binary/object code, e.g., static analysis and code inspection [8, 11]. While it can give an idea on the expected system performance, this technique suffers from the absence of dynamic behavior of the system. The third technique is more accurate and is widely used, although the most time and resource consuming. It relies on executables models/implementations, i.e., high/low-level simulation or execution [37, 41, 49]. These techniques are not exclusive, that is, they can be combined together during the same process

1.3.2 Characterizing ES Performance: Models and Methods

1.3.2.1 Detailed Representations

A widely used technique in the context of ES design is to rely on a detailed description of the target hardware (a virtual platform, an RTL description, and an FPGA implementation) and to perform co-simulation/emulation. That is, to simulate/emulate a model of the application on top of the hardware part of the system [48]. This enables to try different system configurations, e.g., mapping, buffers sizes, and to access a detailed view of the system performance. Besides the issues related to building the detailed hardware description and the application implementation, additional technical issues are to handle such as measuring the performance and managing scalability issues, e.g., limited size of FPGAs. Moreover, an important time is necessary to perform co-simulation/emulation, e.g., simulating the boot of a Linux on an RTL description of a processor takes a half day. Such approach is empirical and provides no guarantee on the obtained performance. The reason is that the simulated/emulated artifact do not generally rely on a clear formal semantics, e.g., a SystemC description of the hardware architecture combined with a C/C^{++} implementation of the application. Hence, only classical testing [4] techniques are possible to check the performance requirements.

1.3.2.2 Abstract Representations

As opposed to the previous procedure, the approach for formal modeling and evaluation of ES performance relies on mathematically sound models of the application and the target hardware. The first advantage of such approach is that it allows to evaluate the system performance early, i.e., before spending too much time building concrete implementations. Furthermore, it enables automatic code generation once the model is validated. The second advantage is that models validation relies on formal verification techniques which provide mathematical guarantees for performance evaluation results (in addition to functional requirements). However, as stated in Sect. 1.2, it is challenging to build detailed hardware models (especially formal ones) during early design stages. Thus, abstract models are often built and combined with the application model in order to assess the expected ES performance formally. This approach requires to characterize the system performance, i.e., to build a parsimonious (abstract) formal representation of performance usually based on incomplete view of the system. Many approximation schemes providing different abstraction and faithfulness levels are possible.

A widely used class of techniques for characterizing performance especially timing uses upper/lower bounds. These techniques are mostly used for modeling and verifying hard real-time systems. They take their roots on well-known theories such as the Queuing theory [22] and the Network Calculus [36]. They essentially reason on best/worst case scenarios and have as objective to build upper/lower bounds on performance. Worst Case Execution Time (WCET) analysis techniques [61], for instance, are used to compute the longest execution time in order to guarantee hard deadlines of executing tasks with respect to some scheduling policy. Such techniques are challenging and heavily dependent on the target hardware. Moreover, they are increasingly difficult and sometime impossible to apply in the case of multi and many-cores architectures [19].

Compositional analysis techniques have been proposed to handle the complexity of performance evaluation of heterogeneous real-time systems. These techniques mainly rely on the Real Time Calculus (RTC) method [56], which characterizes the input stream (workload) and the processing power of some task as arrival and service curves, respectively. It then gives exact bounds on the output stream as a function of its input stream. The latter can then be used as input to the next component and so on. Arrival curves capture upper and lower bounds of arrival time of a class of input events, while service curve captures upper and lower bound of the processing time of consecutive events for any potential stream. RTC was adopted and extended in several researches such as SymTA/S (Symbolic Timing Analysis for Systems) [25], MAST [24], and the Modular Performance Analysis (MPA) [17]. The latter is used within the DOL methodology [57] for system-level performance analysis in the context of ES. As stated earlier, these methods are conceived to be conservative and thus are more appropriate to address stringent constraints such as hard real-time requirements. Consequently, they often result in over-dimensioned designs.

The previous methods generally produce analytical models and have the advantage of short analysis time. However, they generally fail to capture complex interactions and state-dependent behavior. The latter are enabled by another class of models known as operational models, e.g., timed automata [2], which enable a state-space representation of the system. Thus, they allow to build more accurate models. Nevertheless, these suffer from the state-space explosion problem which results in important analysis time. An important advantage, however, is their ability to abstraction. Actually, it is possible using these models to represent the same functionality at different levels of details. Additional techniques for characterizing performance are also used in the literature. They often rely on simple averages or median measures [6, 41]. These provide too coarse characterization and thus are not very useful for building faithful models.

1.3.2.3 Probabilistic Representations

Characterizing performance probabilistically avoids modeling sophisticated hardware functionalities in details. It enables capturing the gist of performance evolution and its associated fluctuation, e.g., the interference due to concurrent access to a shared resource is interpreted as a stochastic evolution over time. This provides a natural view for representing systems performance in a faithful and parsimonious way. For example, when tossing a coin, it is common to consider that its outcome is probabilistic, i.e., head and tail have equal probability to appear when assuming a fair coin. However, this is an abstract view of reality. Actually, if we are able to precisely characterize the coin, e.g., its weight, and it initial position, the environment where the experiment is happening, e.g., wind speed and direction, and the flipping power, we would be able to precisely compute the outcome of the experiment by applying the classical laws of physics.

Probabilistic modeling provides various abstraction possibilities, ranging from simple point estimate with confidence intervals, e.g., mean, variance, to more sophisticated models, e.g., probability density function, Markov Models. This depends not only on the level of details required to solve a given problem but also on the used method to build such models. Several techniques in the literature were extended to a probabilistic setting to take advantage of this view. Such as the Probabilistic WCET analysis techniques [16, 19] which enable to compute probabilistic upper bounds, that is a classical upper bound with the probability to exceed it, probabilistic real-time calculus [29, 53], probabilistic timed automata [32], and stochastic timed automata [5].

1.4 The *ASTROLABE* Approach

In this section, we review a recent work that illustrates our view of probabilistic performance characterization and how it may be used within a complete ES design process. The considered work introduces an approach for building and verifying faithful stochastic ES models that combine the functional behavior of the system with its performance information. We briefly overview the proposed approach and its different steps. Then, we discuss the assumptions it makes for inferring sound probabilistic performance models in order to propose later more general techniques.

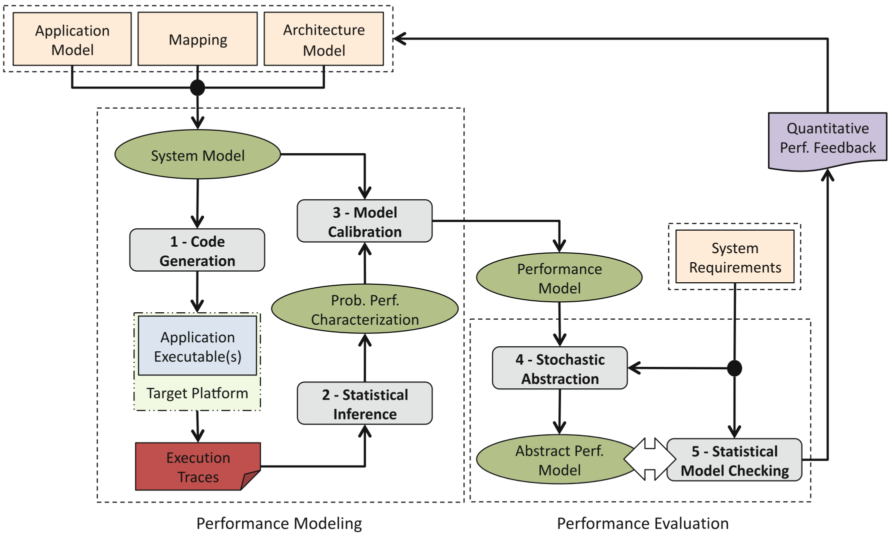


Fig. 1.3 Illustration of the *ASTROLABE* approach

1.4.1 Overview

The *ASTROLABE* approach was introduced in [44]. It targets the earliest phases of systems design and aims at providing fast and accurate quantitative evaluation of possible design alternatives with respect to performance requirements, e.g., time. As shown in Fig. 1.3, the approach considers as inputs a parametric application model, a hardware architecture model, one or more mappings, i.e., a binding of the software into hardware components, and a set of system requirements. The approach starts by automatically building a functional system model consisting of the combination of the software application with the hardware architecture with respect to the given mapping [10]. The input application and architecture models are purely functional and are expressed in the BIP formalism [7]. These may be obtained automatically through refinement from higher level specifications [8] or provided directly by the designer.

A complete iteration of the approach builds and analyzes a specific configuration, where all the application parameters and the mapping are fixed, against the given performance requirements. Quantitative results corresponding to different configurations enable to choose the most adequate and to refine the input models accordingly. To do so, the approach combines two activities, namely system modeling followed by performance evaluation. The former aims at producing a faithful model combining functional and performance information, while the latter aims at performing fast and accurate performance analysis. Both modeling and evaluation are composed of sub-tasks as shown in Fig. 1.3.

The modeling phase consists of three successive steps: (1) A distributed implementation of the application and the corresponding deployment code on the target platform are automatically generated given some runtime support. The code generation step produces instrumented code with respect to the input requirements which specify the performance aspects to characterize. (2) The generated implementation is executed on the target platform and the obtained traces are analyzed to build a probabilistic characterization of the specified performance aspects. This relies on a statistical inference procedure, namely distribution fitting described below. Finally, (3) a model calibration step integrates the obtained probabilistic characterization of performance into the functional system model, which produces a stochastic $\mathcal{S}$BIP model [43, 46].

On the other hand, the performance evaluation phase is twofold: (4) a more compact and equivalent representation with respect to the original model is automatically built [45]. The goal of this step is to speed up the verification of performance requirements. (5) The analysis part is based on Statistical Model Checking algorithms [26, 63], which probabilistically answer if the system model satisfies the performance requirements up to a given precision.

The *ASTROLABE* approach is supported by a tool-flow that automates almost all the underlying steps and it was successfully applied to model and analyze a real-life case-study consisting of a distributed image recognition application running on many-cores platform [44].

1.4.2 Distribution Fitting

The distribution fitting procedure proposed in the *ASTROLABE* approach aims at fitting a probability distribution to performance observations obtained by executing an automatically generated application implementation on a target architecture. It consists of three successive steps which are briefly described in this section (for a complete description kindly refer to [44]).

The first step in this procedure is an exploratory analysis. It consists of verifying that the considered data is adequate for such procedure, i.e., that the performance observations are *independent and identically distributed (iid)*. In addition, this step tries to identify potential probability distribution families that potentially fit the data. The second step in the procedure consists of estimating the parameters of the identified probability distribution candidates. For this, it relies on well-known estimation procedure such as the Maximum Likelihood and the Moment Matching estimates [3, 18]. Finally, once the candidate distributions parameters are estimated, they are evaluated using goodness-of-fit tests to decide if they represent a good fit to the input data and to select the best fit. For this it uses test statistics such as the Kolmogorov–Smirnov (K-S), Anderson–Darling (A-D), and Cramer–Von Mises (C-VM) [31, 60].

1.4.3 Assumptions and Shortcomings

The *ASTROLABE* approach relies on a formal semantics in all its phases, which is important for a consistent design process. The distribution fitting procedure proposed to infer probabilistic models from concrete execution traces of the system represents a fundamental step to ensure a faithful modeling of performance. Furthermore, using abstraction and verification techniques for a quantitative performance evaluation provides fast and accurate analysis results.

Despite these advantages, in its current status, the approach has some weaknesses due to simplifying assumptions and restrictions which are necessary for the performance characterization proposal to work. These assumptions are verified for specific classes of systems where the performance observations satisfy the *iid* assumption. To ensure this assumption, the input models to the approach are restricted to the following:

– The application model is required to follow a process network model of computation, i.e., a set of computation processes (each operating sequentially) coordinated through communication channels. Such models enforce a clear separation between computation and communication.
– The hardware architectures model corresponds to many-cores platforms with homogeneous processing elements sharing a main memory. Processing cores and memory may be organized in clusters, i.e., each cluster has a set of processing elements and a local memory. Clusters may communicate through an NoC or a bus. The considered platforms do not use sophisticated hardware functionalities such as caches, or pipelining. Thus, the main source of interference at this level is the contention on shared resources.
– The considered mappings statically bind each computation processes to a processing core, i.e., no multi-threading on cores is allowed. Communication channels are mapped into shared memory.

1.5 Performance Modeling Using Probabilistic Models

Probabilistic modeling usually consists of manually deriving a probabilistic characterization from given specifications of some artifact. In some cases, we are only given output data of an experiment, e.g., data collected during a poll. Such data represents a partial view of the artifact, that is, in statistical terms, a sample of the complete data population. In such situations, inferential analysis, that is a bottom up path, is used to build the model. Statistical inference is thus the process of probabilistically characterizing the behavior of an artifact based on partial observations. Such approach is often denoted model fitting [35, 60].

In our context we are concerned with performance measurements obtained by simulating/executing embedded software implementations on a hardware architecture. The goal is to characterize the system performance by finding an appropriate model that faithfully and parsimoniously describe it. To do so, various probabilistic models exist, spanning from standard probability distributions, e.g., Exponential or Normal, to more complex ones such as regression models.

From this perspective, data is assumed to be generated by a stochastic process (Definition 1) for which the governing law is unknown. The goal of the statistical inference is to infer such a law from given observations which represent a partial view of the process since the whole population is generally not available. Formally, given $x_1, \ldots, x_n$ a set of observations, there exist $X_1, \ldots, X_n$ random variables such that x_i is a possible realization of X_i, for $1 \leq i \leq n$. The set of random variables $X_1, \ldots, X_n$ represent together a stochastic process.

Definition 1 (Stochastic Process). Given two sets S and T, a stochastic process $\mathcal{X}$ is a collection of random variables $\{X_i \mid i \in T\}$ where each X_i is an S-valued random variable. The set S is called the state space of the process.

In the context of performance modeling, the random variables $X_1, \ldots, X_n$ represent measurements of the same performance metric, say the execution time of a computation process, at different time points, i.e., the indexes $i \in T$ actually represent the measurement times. They typically correspond to the set of non-negative integers $\mathbb{Z}^*$ in the case of discrete-time stochastic processes and to the set of non-negative real number $[0, \infty)$ for continuous-time stochastic processes.

Depending on the target probabilistic model and on the underlying data, different inference methods can be used to construct the desired model. In the *ASTROLABE* approach, for instance, we targeted a simple probabilistic model consisting of standard probability distributions as discussed in the previous section. In that work, it is assumed that performance observations are *independent and identically distributed*. *Independent* means that given the realization x_i of the random variable X_i, it does not provide any information about the realization x_j of X_j, for $i < j \leq n$. Formally $P(X_j = x_j \mid X_i = x_i) = P(X_j = x_j)$. *Identically distributed* states that the underlying random variables follow a same probability distribution $D(\omega)$, where $\omega \in \Theta$ is the set of parameters of the distribution defined over the space Θ.

This *iid* assumption may be seen as an abstraction choice. That is, since the underlying random variables are independent and they follow the same distribution $D(\omega)$, they can be modeled by a single random variable X that follows the distribution $D(\omega)$ and where $x_i, \ldots, x_n$ are possible realizations of X with probabilities defined by $D(\omega)$. In this view, one can forget about the time dimension of the stochastic process, i.e., the measurements order, as it is assumed to have no impact on the process evolution. While providing a nice abstract and parsimonious model of performance, the *iid* assumption is usually too strong. In practice, measurements performed on real-life systems relying on sophisticated hardware functionalities do not generally satisfy it.

Consider a hardware architecture implementing functionalities such as the ones presented in Sect. 1.2. Such hardware introduces a dependency between the random variables $X_1, \ldots, X_n$ representing the performance metric. Concretely, we can write that $P(X_i = x_i) = P(X_i = x_i \mid X_{i-1} = x_{i-1}, \ldots, X_0 = x_0)$. Assume we are measuring the execution time of some computation process at different times.

The value obtained at time *i* may impact future measures because of the effects of caches, branch prediction, or pipelining. Such hardware components are known to introduce a history within the data evolution [62]. In general, they may create additional dependency with other random variables modeling the execution context. For instance, the execution time of a process depends on the data cache status (hit or miss), the number of concurrent processes accessing a shared resource, the used arbitration policy, and other factors. Moreover, the impact of such components is not always the same, e.g., a cache hit does not systematically imply a lower execution time. This kind of behavior is known as timing anomaly in the literature [52].

Capturing such a history or dependency is important to characterize performance faithfully. However, the obtained models must be parsimonious and concise to enable fast analysis. Ideally, they should be obtained automatically. In the following we first present some models that we believe promising for ensuring such requirements. Next, we survey some techniques that enable to obtain such models automatically from measurements.

1.5.1 Probabilistic Models

The distribution fitting procedure presented in [44] is one among various model fitting procedures [35, 60]. It aims at inferring a probability distribution as a target performance model. Different probabilistic models may be used and they imply to use different techniques with different complexities. The chosen model may or may not take into account the internal structure of the data induced by the hardware part of the system. This primarily depends not only on the required level of abstraction, but also on the inherent properties of the data. In the following, we present three models that capture the data internal structure in different ways.

1.5.1.1 Mixture Distributions

Mixture distributions [40, 47] are useful when the considered data provides clues to be drawn from different parent populations. In practice this may corresponds to performance measurements performed in different computation modes, e.g., the energy consumption measured during a power-save mode and a normal mode. The decision of fitting such a model to the data usually follows an exploratory analysis (similar to the distribution fitting procedure in [44]). For instance, a histogram of the data may show a clear multi-modality[5] as illustrated in Fig. 1.4 which shows a bimodal shape.

A mixture distribution enables to capture data sub-populations and to fit it with a single probability distribution that can be used to perform classical computation,

[5]A mode can be identified as a prominent peak in the histogram.

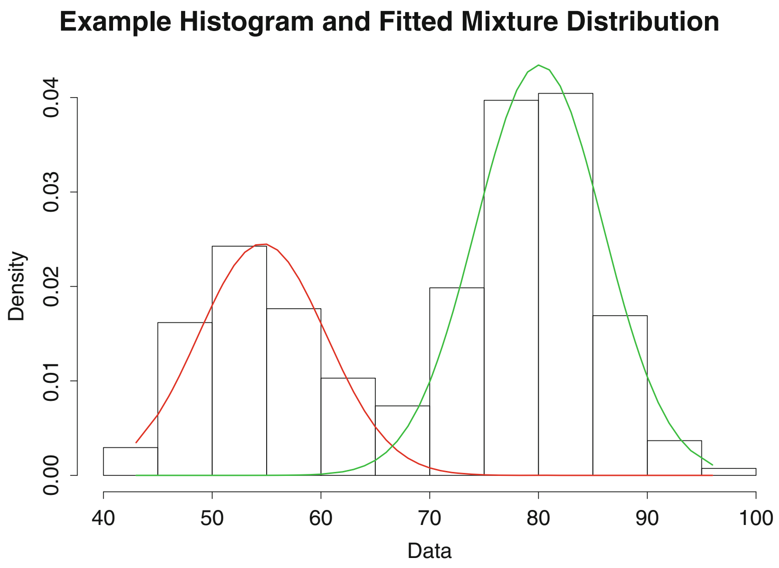


Fig. 1.4 A mixture distribution example

e.g., expectations. Let $f_1(x), \ldots, f_m(x)$ be a set of probability density functions associated with distributions $D_1(\omega), \ldots, D_m(\omega)$, and let $w_1, \ldots, w_m$ be some weights such that $\sum_{i=1}^{m} w_i = 1$ (they follow an additional probability distribution which we denote the selection distribution W). Then, the mixture distribution density function f is defined as $\sum_{i=1}^{m} w_i f_i$. An important assumption in this context is the independence between the selection distribution W and the underlying distributions f_i. That is, we assume that a component distribution is first selected randomly according to W, then the selected distribution is used to draw a value. Figure 1.4 shows an example data illustrated using a histogram with a clear bimodality. The figure also shows the fitted mixture distribution composed of two Normal distributions [9]. It is worth mentioning that we are not aware of works that uses mixture distributions to characterize performance of ES.

1.5.1.2 Regression Models

Regression models provide a mathematical tool to understand, explain, and forecast on data. They enable to model the underlying stochastic process evolution as depending on internal or external factors. Hence, they allow for capturing dependencies within the process or with respect to other impacting factors. More precisely, the process evolution (denoted response or output variable) can be explained by its own history evolution or by other random variables (denoted explaining variables). In our context, we focus on the first case, that is we provide models that characterize the process evolution in terms of its own history. The second case is equally interesting for investigation in this context as it enables to characterize a given performance metric as a function of its execution context, e.g., caches status, shared resource contention.

In general, regression models can be linear (respectively, non-linear), that is, the response variable is a linear (respectively, non-linear) combination of the explaining variables. Furthermore, different regression models can be used to model stochastic processes with different properties. In the following, we focus on regression models for stationary stochastic processes [13, 42]. That is the process has a constant variance over time (no trend), a constant autocorrelation structure over time, and no periodic fluctuation (no seasonal behavior). We believe that this class of process may be interesting for ES performance measurements.

There exist various regression models that can be used to characterize stationary processes. A well-known class is the *Autoregressive Models* (AR) which are linear regression models that represent the response variable as a linear function of previous steps (explaining variables) in addition to a random error.

$$X_i = c + \theta_1 X_{i-1} + \cdots + \theta_p X_{i-p} + \epsilon_i = c + \epsilon_i + \sum_{j=1}^{p} \theta_i X_{j-1}$$

where c is a constant, $\theta_i, \ldots, \theta_p$ are parameters of the model, and p is the order of the AR model, i.e., the number of considered explaining variables (how many previous steps impact the next one). Another commonly used regression model for stationary processes is the *Moving Average* (MA). The latter is also a linear regression model. However, it represents the response variable as a function of random error terms evolving around the expected value (the mean) of the data.

$$X_i = \mu + \epsilon_i + \varphi_1 \epsilon_{i-1} + \cdots + \varphi_q \epsilon_{i-q} = \mu + \epsilon_i + \sum_{k=1}^{q} \varphi_i \epsilon_{j-1}$$

where μ is the expectation of X_i, $\varphi_i, \ldots, \varphi_q$ are parameters of the model, and q is the order of the model. In both models, ϵ represent random error terms which are often assumed to be independent and identically distributed random variables following a Normal distribution $N(0, \sigma^2)$, where σ^2 is its variance. In addition to the previous models a combination of them can also be used to characterize stationary stochastic processes. This is known as Box–Jenkins ARMA model [12, 13] and includes p autoregressive terms and q moving average ones.

$$X_i = c + \epsilon_i + \sum_{j=1}^{p} \theta_i X_{j-1} + \sum_{k=1}^{q} \varphi_i \epsilon_{j-1}$$

It is also possible to handle nonstationary processes by differencing the process one or more times to achieve stationarity. This leads to the so-called ARIMA models, where the “I” stands for integrated [42].

A few works in the literature have used regression models to characterize performance of embedded systems. For example, in [21], regression models corresponding to processes execution time over a single processor are used to calibrate high-level behavioral models in the context of the VCC system-level modeling and analysis methodology. In [1], a tool for power estimation on co-processors is presented, it also relies on regression techniques to learn predictive power consumption models as functions of different aspects of the system.

1.5.1.3 Markov Models

Markov models or chains are stochastic processes that have a particular properties known as the Markov property. Concretely, they are memoryless processes, in the sense that the probability of reaching the next state of the system only depends on its current state and not on the complete history of the process. Formally, we write that:

$$P(X_{i+1} = x_{i+1} \mid X_i = x_i, \ldots, X_0 = x_0) = P(X_{i+1} = x_{i+1} \mid X_i = x_i)$$

A Markov chain is often represented as a directed graph where vertices represent the model states and the edges represent transitions from one state to another. An edge from a state s_i to a state s_{i+1} is labeled with the probability $P(X_{i+1} = x_{i+1} \mid X_i = x_i)$. Figure 1.5 illustrates an example of a Markov chain modeling the behavior of the Craps Gambling game [4]. In this game, a player starts by rolling two fair six-sided dice. The outcome of the two dice determines whether he wins or not. If the outcome is 7 or 11, the player wins. If the outcome is 2, 3, or 12, the player looses. Otherwise, the dice are rolled again taking into account the previous outcome (called point). If the new outcome is 7, the player looses. If it is equal to point, he wins. For any other outcome, the dice are rolled again and the process continues until the player wins or loses.

Markov models are by far the most used to characterize ES performance [14, 32, 33]. However, most of these works start from systems specifications to manually build the Markov model. Only few works follow an inferential path, i.e., uses concrete execution data to obtain a faithful model [38, 39].

1.5.2 Learning and Fitting Techniques

In this section we survey some techniques that may be used to automatically build the models presented above. These techniques mainly rely on model fitting which is represent an important part of the family of machine learning techniques. The latter is an active field of research and learning algorithms are constantly developed and improved in order to address new challenges and new classes of problems (see [59] for a recent survey).

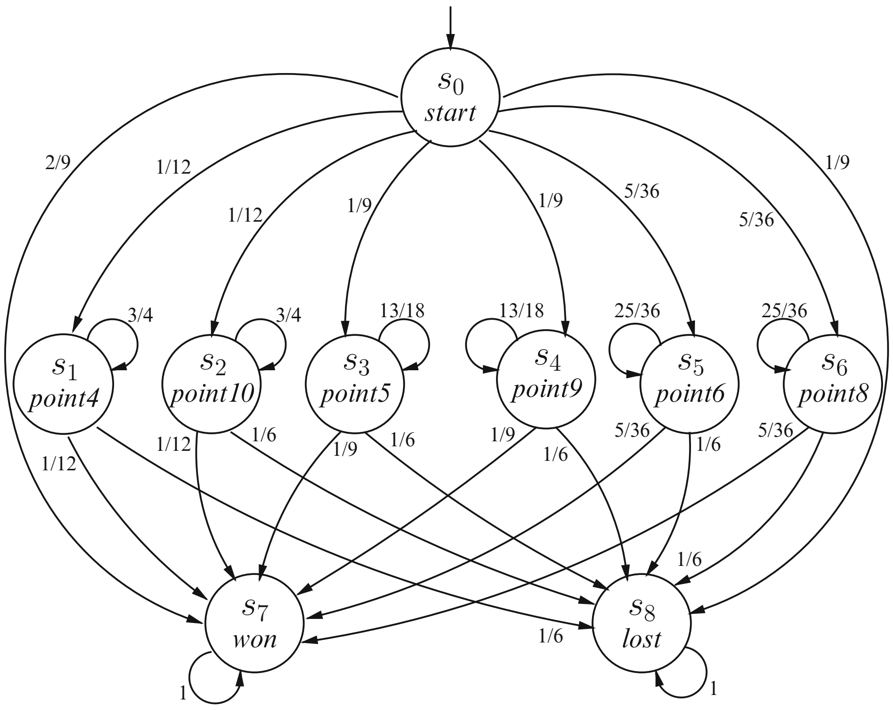


Fig. 1.5 A Markov chain example

1.5.2.1 Fitting Mixture Models

To estimate the parameters of mixture models, i.e., the parameters ω of the sub-distributions D_i and the selection distribution W, one may use classical estimation techniques such as the likelihood estimate or the moment matching estimate [3, 18]. However, in this setting such techniques often fail to perform the estimation. Generally, for fitting a mixture model to input data, we rely on a well-known algorithm, namely the Expectation Maximization (EM) [13, 40]. This is an iterative algorithm used for computing the maximum likelihood estimate when the considered data is incomplete. Some tools also exist for automatically fitting such models [9].

1.5.2.2 Fitting Regression Models

Fitting an ARMA model to the data can be done using classical least square regression after fixing the orders p and q [42]. This will estimate the values of the model parameters by minimizing the error term. The fitting procedure follows usually three main steps, namely model identification, that is to check if the considered data is stationary and if there is a seasonality that need to be taken into account (one can use run plot or autocorrelation plot to identify it). This first step is also useful to identify p and q. The second step is the model estimation, that is estimating the parameters of the model. As stated earlier one can relies on classical estimation approaches such as least squares and maximum of likelihood [3, 18]. The last step of the procedure is an evaluation step. It consists of verifying that the model residuals are independent drawings from a fixed distribution [42]. Recently, a new and rich class of models and techniques were proposed in this context. They are known state-space models and were originally developed in the context of linear systems control [13]. They rely, for instance, on Kalman recursions and EM algorithm [13].

1.5.2.3 Learning Markov Models

In the last years, several works were proposed to learn Markov Models with their different varieties. Aalergia [38] is a state merging algorithm that enable to learn deterministic Markov Chain. It is a variant of the alergia algorithm proposed by Carrasco and Oncina in the early 1990s [15]. Given a sample of execution traces, the algorithm builds an intermediate graph that represents all the traces in the input sample and their corresponding frequencies. Then, it iteratively merges the nodes of the graph that have the same labels and similar probability distributions until reaching a compact version, which corresponds to the final Markov chain. An extension of this algorithm was also proposed in [39] to enable learning Markov models with non-determinism, namely Markov Decision Processes (MDPs). Other techniques based on the Bayesian approach were proposed for learning the parameters and the structure of Hidden Markov Models (HMMs) [51]. They are also following a state merging procedure [55]. The previously mentioned works essentially focus on discrete-time models. Other works, albeit few, exist for learning continuous-time Markov models. For instance, in [54] it is proposed to learn continuous-time Markov chains (CTMCs) models from sample traces following the state merging approach by providing a variant of the alergia algorithm. In [20], an algorithm that follows the same strategy is proposed to learn more general continuous-time models, namely General Semi-Markov Processes (GSMPs).

1.5.3 Perspectives

In the previous sections, we focused on presenting models and techniques for characterizing performance probabilistically. Nonetheless, given such characterization, an important question remains on how to combine the performance characterization with the function behavior of the system within a single formal framework. Within the *ASTROLABE* approach, a calibration procedure is used in order to augment functional BIP models with performance characterization in form of probability distributions. This procedure produces stochastic $\mathcal{S}$BIP models [43, 46].

We speculate that a similar procedure is potentially applicable when charactering performance as mixture distributions since they are formally described as probability distributions. On the other side, this might be more complicated for regression and Markov models. The latter might be easier to integrate within BIP models due to their underlying operational semantics. In [43], a transformation of a Markov chain to a stochastic $\mathcal{S}$BIP component is formally defined. However, integrating it within the functional model requires to add the appropriate glue. Regression models are more involved because of their underlying analytical models. They require more investigation as how to combine them with operational functional models, e.g., BIP, and about the semantics it will induce.

References

1. S. Ahuja, D.A. Mathaikutty, A. Lakshminarayana, S.K. Shukla, Scope: statistical regression based power models for co-processors power estimation. J. Low Power Electron. **5**(4), 407–415 (2009)
2. R. Alur, D.L. Dill, A theory of timed automata. Theor. Comput. Sci. **126**(2), 183–235 (1994)
3. T.W. Anderson, J.D. Finn, *The New Statistical Analysis of Data* (Springer, New York, 1996)
4. C. Baier, J.P. Katoen, *Principles of Model Checking* (MIT Press, Cambridge, MA, 2008)
5. C. Baier, N. Bertrand, P. Bouyer, T. Brihaye, M. Grösser, Probabilistic and topological semantics for timed automata, in *Proceedings of the 27th International Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS* (Springer, Berlin/Heidelberg, 2007), pp. 179–191
6. A. Bakshi, V.K. Prasanna, A. Ledeczi, MILAN: a model based integrated simulation framework for design of embedded systems. ACM Sigplan Not. **36**(8), 82–93 (2001)
7. A. Basu, B. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen, J. Sifakis, Rigorous component-based system design using the BIP framework. IEEE Softw. **28**(3), 41–48 (2011)
8. A. Basu, S. Bensalem, M. Bozga, P. Bourgos, M. Maheshwari, J. Sifakis, Component assemblies in the context of manycore, in *Formal Methods for Components and Objects* (Springer, New York, 2013), pp. 314–333
9. T. Benaglia, D. Chauveau, D.R. Hunter, D.S. Young, Mixtools: an R package for analyzing finite mixture models. J. Stat. Softw. **32**(6), 1–29 (2009)
10. P. Bourgos, Rigorous design flow for programming manycore platforms. Ph.D. thesis, Grenoble University, 2013
11. P. Bourgos, A. Basu, M. Bozga, S. Bensalem, J. Sifakis, K. Huang, Rigorous system level modeling and analysis of mixed HW/SW systems, in *MEMOCODE* (2011), pp. 11–20
12. G.E.P. Box, G.M. Jenkins, G.C. Reinsel, *Time Series Analysis: Forecasting and Control*. Forecasting and Control Series (Prentice Hall, Englewood Cliffs, 1994)
13. P.J. Brockwell, R.A. Davis, *Introduction to Time Series and Forecasting*. Number v. 1 in Introduction to Time Series and Forecasting (Springer, New York, 2002)
14. P.E. Bulychev, A. David, K.G. Larsen, M. Mikucionis, D.B. Poulsen, A. Legay, Z. Wang, UPPAAL-SMC: statistical model checking for priced timed automata, in *Proceedings 10th Workshop on Quantitative Aspects of Programming Languages and Systems, QAPL 2012, Tallinn, 31 March and 1 April 2012* (2012), pp. 1–16
15. R.C. Carrasco, J. Oncina, Learning stochastic regular grammars by means of a state merging method, in *International Colloquium on Grammatical Inference* (1994), pp. 139–152
16. F.J. Cazorla, E. Quinones, T. Vardanega, L. Cucu-Grosjean, B. Triquet, G. Bernat, E. Berger, J. Abella, F. Wartel, M. Houston, L. Santinelli, L. Kosmidis, C. Lo, D. Maxim, PROARTIS: probabilistically analysable real-time systems, Research Report RR-7869, INRIA, 2012

17. S. Chakraborty, S. Künzli, L. Thiele, A general framework for analysing system properties in platform-based embedded system designs, in *Design Automation and Test in Europe*, Citeseer, vol. 3 (2003), p. 10190
18. G. Cowan, *Statistical Data Analysis* (Oxford University Press, Oxford, 1998)
19. L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzeti, E. Quinones, F.J. Cazorla, Measurement-based probabilistic timing analysis for multi-path programs, in *The 24th Euromicro Conference on Real-Time Systems*, Pisa, Italy (2012)
20. A. de Matos Pedro, P.A. Crocker, S.M. de Sousa, Learning stochastic timed automata from sample executions, in *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change* (Springer, New York, 2012), pp. 508–523
21. P. Giusto, G. Martin, E. Harcourt, Reliable estimation of execution time of embedded software, in *Proceedings of the Conference on Design, Automation and Test in Europe, DATE '01* (IEEE Press, Piscataway, NJ, USA, 2001), pp. 580–589
22. D. Gross, J.F. Shortle, J.M. Thompson, C.M. Harris, *Fundamentals of Queueing Theory*. Wiley Series in Probability and Statistics (Wiley, New York, 2011)
23. W. Haid, M. Keller, K. Huang, I. Bacivarov, L. Thiele, Generation and calibration of compositional performance analysis models for multi-processor systems, in *ICSAMOS* (2009), pp. 92–99
24. M.G. Harbour, J.J.G. García, J.C.P. Gutiérrez, J.M.D. Moyano, Mast: modeling and analysis suite for real time applications, in *13th Euromicro Conference on Real-Time Systems* (IEEE, Computer Society, Washington, DC, USA, 2001), pp. 125–134
25. R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, R. Ernst, System level performance analysis - the SymTA/S approach, in *IEEE Proceedings Computers and Digital Techniques* (2005)
26. T. Hérault, R. Lassaigne, F. Magniette, S. Peyronnet, Approximate probabilistic model checking, in *Verification, Model Checking, and Abstract Interpretation* (2004), pp. 73–84
27. K. Huang, W. Haid, I. Bacivarov, M. Keller, L. Thiele, Embedding formal performance analysis into the design cycle of MPSoCs for real-time streaming applications. ACM Trans. Embed. Comput. Syst. **11**(1), 8:1–8:23 (2012)
28. Z.J. Jia, A. Núñez, T. Bautista, A.D. Pimentel, A two-phase design space exploration strategy for system-level real-time application mapping onto MPSoC. Microprocess. Microsyst. **38**(1), 9–21 (2014)
29. Y. Jiang, Y. Liu, *Stochastic Network Calculus* (Springer, London, 2008)
30. K. Keutzer, S. Malik, S. Member, A.R. Newton, J.M. Rabaey, A. Sangiovanni-vincentelli, System-level design: orthogonalization of concerns and platform-based design. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **19**, 1523–1543 (2000)
31. S.A. Klugman, H.H. Panjer, G.E. Willmot, *Loss Models: From Data to Decisions*, vol. 715 (Wiley, New York, 2012)
32. M.Z. Kwiatkowska, G. Norman, R. Segala, J. Sproston, Automatic verification of real-time systems with discrete probability distributions. Theor. Comput. Sci. **282**(1), 101–150 (2002)
33. M. Kwiatkowska, G. Norman, D. Parker, Probabilistic verification of Herman's self-stabilisation algorithm. Form. Asp. Comput. **24**(4), 661–670 (2012)
34. K. Lampka, G. Giannopoulou, R. Pellizzoni, Z. Wu, N. Stoimenov, A formal approach to the WCRT analysis of multicore systems with memory contention under phase-structured task sets. Real-Time Syst. **50**(5–6), 736–773 (2014)
35. J.-Y. Le Boudec, *Performance Evaluation of Computer and Communication Systems* (EPFL Press, Lausanne, 2010)
36. J.-Y. Le Boudec, P. Thiran, *Network Calculus: a Theory of Deterministic Queuing Systems for the Internet* (Springer, Berlin/Heidelberg, 2001)
37. P. Lieverse, P. Van Der Wolf, K. Vissers, E. Deprettere, A methodology for architecture exploration of heterogeneous signal processing systems. J. VLSI Sig. Process. Syst. Sig. Image Video Technol. **29**(3), 197–207 (2001)

38. H. Mao, Y. Chen, M. Jaeger, T.D. Nielsen, K.G. Larsen, B. Nielsen, Learning probabilistic automata for model checking, in *QEST* (2011), pp. 111–120
39. H. Mao, Y. Chen, M. Jaeger, T.D. Nielsen, K.G. Larsen, B. Nielsen, Learning Markov decision processes for model checking. arXiv preprint arXiv:1212.3873 (2012)
40. N. Matloff, *From Algorithms to Z-Scores: Probabilistic and Statistical Modeling in Computer Science* (University Press of Florida, Gainesville, 2009)
41. S. Mohanty, V.K. Prasanna, Rapid system-level performance evaluation and optimization for application mapping onto SoC architectures, in *ASIC/SOC Conference, 2002. 15th Annual IEEE International* (IEEE, Piscataway, NJ, USA, 2002), pp. 160–167
42. NIST/SEMATECH, *NIST/SEMATECH e-Handbook of Statistical Methods* (2016)
43. A. Nouri, Rigorous system-level modeling and performance evaluation for embedded system design, Theses, Université Grenoble Alpes, 2015
44. A. Nouri, M. Bozga, A. Molnos, A. Legay, S. Bensalem, Building faithful high-level models and performance evaluation of manycore embedded systems, in *Twelfth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE), 2014* (IEEE, 2014), pp. 209–218
45. A. Nouri, B. Raman, M. Bozga, A. Legay, S. Bensalem, Faster statistical model checking by means of abstraction and learning, in *Proceedings of Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, 22–25 Sept 2014* (2014), pp. 340–355
46. A. Nouri, S. Bensalem, M. Bozga, B. Delahaye, C. Jégourel, A. Legay, Statistical model checking QoS properties of systems with SBIP. Softw. Tools Technol. Trans. **17**(2), 171–185 (2015)
47. R. Pearson, *Exploring Data in Engineering, the Sciences, and Medicine* (Oxford University Press, New York, 2011)
48. A.D. Pimentel, The artemis workbench for system-level performance evaluation of embedded systems. Int. J. Embed. Syst. **3**, 181–196 (2008)
49. A.D. Pimentel, C. Erbas, S. Polstra, A systematic approach to exploring embedded system architectures at multiple abstraction levels. Comput. IEEE Trans. **55**(2), 99–112 (2006)
50. A.D. Pimentel, M. Thompson, S. Polstra, C. Erbas, Calibration of abstract performance models for system-level design space exploration. J. Sig. Process. Syst. **50**(2), 99–114 (2008)
51. L. Rabiner, A tutorial on hidden Markov models and selected applications in speech recognition. Proc. IEEE **77**(2), 257–286 (1989)
52. J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, B. Becker, A definition and classification of timing anomalies, in *6th International Workshop on Worst-Case Execution Time (WCET), Analysis, 4 Jul 2006, Dresden, Germany* (2006)
53. L. Santinelli, L. Cucu-Grosjean, Toward probabilistic real-time calculus. SIGBED Rev. **8**(1), 54–61 (2011)
54. K. Sen, M. Viswanathan, G. Agha, Learning continuous time Markov chains from sample executions, in *First International Conference on the Quantitative Evaluation of Systems QEST.* (IEEE, Computer Society, Washington, DC, USA, 2004), pp. 146–155
55. A. Stolcke, S. Omohundro, Hidden Markov model induction by Bayesian model merging, in *Advances in Neural Information Processing Systems* (1993), pp. 11–11
56. L. Thiele, S. Chakraborty, M. Naedele, Real-time calculus for scheduling hard real-time systems, in *International Symposium on Computer Architecture*, vol. 4 (2000), pp. 101–104
57. L. Thiele, I. Bacivarov, W. Haid, K. Huang, Mapping applications to tiled multiprocessor embedded systems, in *Application of Concurrency to System Design* (2007)
58. L. Thiele, L. Schor, I. Bacivarov, H. Yang, Predictability for timing and temperature in multiprocessor system-on-chip platforms, in *ACM Transactions on Embedded Computing Systems (TECS) - Special Section on ESTIMedia12, LCTES11, Rigorous Embedded Systems Design, and Multiprocessor*, 12 March 2013
59. S. Verwer, R. Eyraud, C. de la Higuera, Results of the pautomac probabilistic automaton learning competition, in *International Conference on Grammatical Inference* (2012), pp. 243–248
60. D. Vose, *Risk Analysis: a Quantitative Guide* (Wiley, New York, 2008)

61. R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, P. Stenström, The worst-case execution-time problem—overview of methods and survey of tools. ACM Trans. Embed. Comput. Syst. **7**(3), 36:1–36:53 (2008)
62. R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, C. Ferdinand, Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. IEEE Trans. CAD Integr. Circuits Syst. **28**(7), 966–978 (2009)
63. H.L.S. Younes, Verification and planning for stochastic processes with asynchronous events. Ph.D. thesis, Carnegie Mellon, 2005

Chapter 2
Resource-Driven Modelling for Managing Model Fidelity

Ashur Rafiev, Andrey Mokhov, Fei Xia, Alexei Iliasov, Rem Gensh, Ali Aalsaud, Alexander Romanovsky, and Alex Yakovlev

2.1 Introduction

Systems with large scale concurrency and complexity, e.g. computation systems built upon architectures with multiple and increasingly many processing cores with heterogeneity among the components, are becoming more popular and commonplace [6]. The hardware motivations are clear, as concurrency scaling can help delay the potential saturation of Moore's Law with current and future CMOS technology and better use the opportunities provided by the technology scaling. In this environment, software designs are increasingly focused towards greater concurrency and mapping to such many-core hardware [16].

If we regard elements of computation, such as software, hardware, energy, time, etc. as resources, then computation itself can be regarded as behaviours of the entire resource space. Existing modelling approaches usually include some degree of representation of resources [5]. Functional units such as transistors, gates, CPUs, memory, software threads, etc. need to be represented in functional models, and non-functional parameters including power, time, temperature, etc. also need to be represented in models that are used to study non-functional behaviours of systems. Existing modelling methods in the literature therefore embed certain degrees of resource representation. However, a modelling method that is entirely based on representing resources and their dependencies has, to our knowledge, not yet been investigated. Being able to reason about resources and their interdependency during computation directly, on the other hand, should be advantageous for studying the

A. Rafiev (✉) • A. Mokhov • F. Xia • A. Iliasov • R. Gensh • A. Aalsaud • A. Romanovsky • A. Yakovlev
Newcastle University, Newcastle upon Tyne, UK
e-mail: ashur.rafiev@ncl.ac.uk; andrey.mokhov@ncl.ac.uk; fei.xia@ncl.ac.uk; alexei.iliasov@ncl.ac.uk; r.gensh@ncl.ac.uk; a.m.m.aalsaud@ncl.ac.uk; alexander.romanovsky@ncl.ac.uk; alex.yakovlev@ncl.ac.uk

A. Molnos, C. Fabre (eds.), *Model-Implementation Fidelity in Cyber Physical System Design*, DOI 10.1007/978-3-319-47307-9_2

more complex systems of today and the future. For instance, power consumption has become a crucial limiting factor for the continued expansion of the world's computing capabilities and power is one of the most straightforward resources [11].

Functional resources, such as hardware and software in large complex systems, tend to form hierarchical structures, for instance, the levels of detail in hardware include the entire spectrum from transistors to gates to function blocks to entire CPUs to multiple CPUs with supporting logic, memory, etc. For system designers, software (e.g. applications), operating systems and the platforms on which these are run also form natural design layers with clear boundaries between the layers. Such structures are usually conveniently modelled with traditional hierarchical modelling methods, with the modelling levels of abstraction corresponding to these system layers of concern [15].

This is, however, not always optimal for analysis, design and runtime management. In most cases these require the modelling of particular parameters and the "modelling fidelity" [21] should, ideally, be determined by the parameter(s) under study [28]. For instance, when a part of a system makes a crucial contribution to the power consumption of the entire system and small changes may have a significant effect, it pays to study it in detail, i.e. at some lower layer of abstraction. On the other hand, to moderate the modelling, analysis and design effort, and potentially runtime overhead for models that need to be used in runtime, other less significant parts of the system should be studied at higher levels of abstraction. When this "centre of gravity" of system operation concerning power can dynamically move around the system, traditional hierarchical modelling methods are ill positioned for efficient representation.

Hierarchical methods, because of their complexity, are usually less straightforward to use than flat representations. Petri nets [3], which exemplify flat modelling methods, have extremely simple semantics and offer conveniences in reasoning, proofing and other aspects of analysis, a quality shared by other flat modelling methods. But when the modelling needs span multiple layers in a hierarchy it becomes somewhat difficult to adopt flat methods as study tools.

In this chapter, we present a systematic resource-driven approach to modelling that emphasises resources and dependencies between resources. The distinction between static design-time modelling which focus on types or classes of resources and run-time dynamic modelling which focuses on actual instances of resources will be made, with the proposed approach covering both issues. The semantics and other theoretical details will be presented, and derived modelling techniques, including the simulation tool ArchOn, will be used to solve real-world problems. For instance, scalable simulations and finding the optimal scaling factor for homogeneous many-core systems considering performance, energy and reliability (PER) trade-offs will be shown to be potential application areas of the method. This resource-driven focus will further lead to the presentation of a method of achieving resource parameter-proportional fidelity in models through analysing abstraction hierarchies and using cross-layer cuts. And this will be demonstrated through solving real PER and system task scheduling trade-offs in heterogeneous multi-core systems.

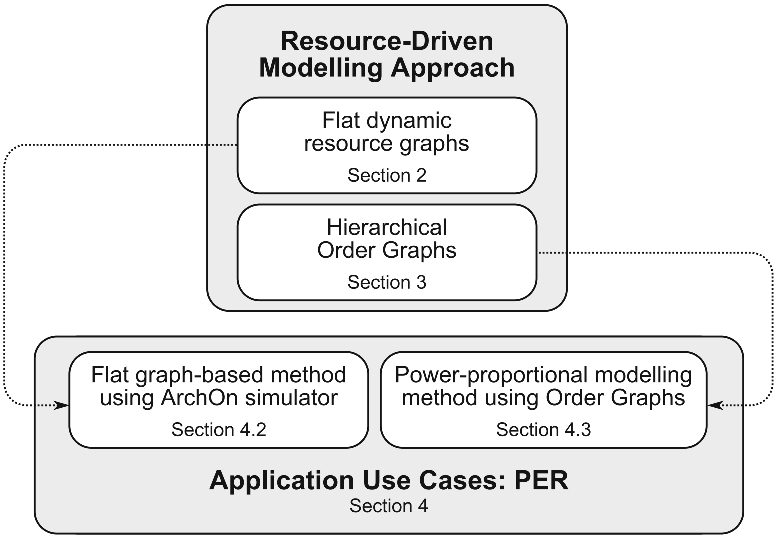

Resource-Driven Modelling Approach
Flat dynamic resource graphs
Section 2
Hierarchical Order Graphs
Section 3
Flat graph-based method using ArchOn simulator
Section 4.2
Power-proportional modelling method using Order Graphs
Section 4.3
Application Use Cases: PER
Section 4

Fig. 2.1 The organisation of the chapter

The rest of this chapter is organised as shown in Fig. 2.1. Section 2.2 introduces the resource-driven modelling approach in a form of static and dynamic resource graphs, and also defines the notions of implementability and resource quantification. Section 2.3 describes Order Graphs (OG)—a hierarchical modelling formalism, and the concept of cross-layer cuts. In Sect. 2.4, methods derived from these theories are used solve the real-life problem of performance, energy and reliability interplay in systems. Section 2.5 concludes the chapter.

2.2 Resource-Driven Modelling

The central subject of our method is the study of a computational platform comprising a number of diverse resources and the way resources may be handled in order to realise a computation [24]. A resource is in this case an indivisible element required by the system in order to change its state, and it is defined by its function and availability in relation to this transition. With the word "resources" we make the point that we do not exclude computation, communication, or other facilities, e.g. energy and time.

We propose to represent a system with a relation graph (R, D), consisting of a set of vertices R and a set of edges $D \subseteq R \times R$. Each vertex $r \in R$ represents a single resource and each edge $d = \langle r_1, r_2 \rangle \in D$ represents a dependency between two resources $r_1, r_2 \in R$. Modelling different types of resources may be achieved by labelling the graph, as illustrated in Fig. 2.2a.

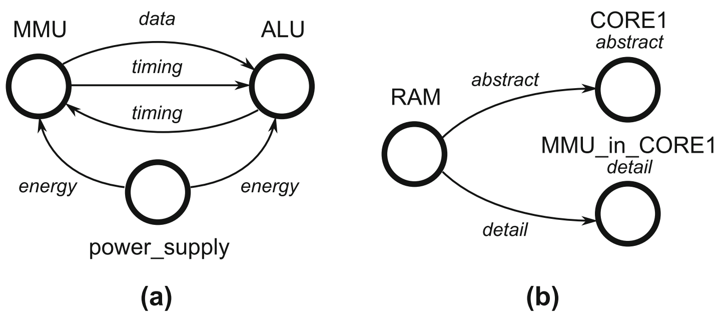


Fig. 2.2 Examples of using flat labelled graphs to reason about diverse resource types and dependencies (**a**), and different levels of abstraction (**b**)

Organising systems, both practically and conceptually, as hierarchies is a popular way of thinking and engineering. The practical motivation for this is manageability. This is the “natural” way for humans to reason about, design and organise most of our systems. In Sect. 2.3 we emphasise the hierarchy-based cross-layer aspects of our work, while this section focuses on the flat graph models.

Figure 2.2b demonstrates that the flat labelled graph approach can, in fact, facilitate the cross-layer way of thinking as long as the behaviour of different layers of abstraction is coherent. A label can be viewed as a condition that includes or excludes an edge or a vertex, giving a graph *projection* onto that label. The complexity of the system can be dealt with using projections of the resource graphs. With resources as diverse as a software instruction or a single hardware gate, within the same single graph executed in a transition, we could reason about different parts of the system at different abstraction layers. This helps a designer focus their attention on any particular details of a system they want, and build a system either top down or bottom up or with mixed-level components at different stages of development.

At the same time, this does not prevent designers to isolate concerns and concentrate on some layers only. For instance, all resources in one transition could be elements of the same layer, or a software engineer could arrange complex low-level software resources for detailed study with coarse-grain hardware resources provided by hardware colleagues (which are not the specific target of concern) in the same transition.

2.2.1 System Design and Implementation

This section introduces the basic concepts of the resource-driven modelling and draws the boundary between the static and dynamic resource representations of a system. It is essential to understand that the knowledge of a system (its definition or design) exists independently of its implementation. In order for the system to be realised, we need to map this knowledge onto a set of resources—an architecture. Thence, if the architecture contains enough resources of the right type, the system implementation or implementations may emerge.

An unconstrained *architecture* $\mathcal{A}^+ = (R, T^r, \tau^r)$ is the set of discrete resources R and the set of resource types T^r, where $\tau^r : R \to T^r$ is the resource type assignment function. Such an architecture is called unconstrained because it does not put any restriction on how the resources may interact. Constrained architectures will be introduced later in Sect. 2.2.4. For now, we see an architecture as a "soup" of resources, where every resource is equally accessible.

The *system design* $\mathcal{S} = (V, E, T^v, \tau^v)$ is a graph, where V is the set of vertices, $E \subseteq V \times V$ is the set of edges (dependencies), T^v is the set of vertex types, and $\tau_v : V \to T^v$ is the type assignment for vertices. The type of some edge $\langle v_1, v_2 \rangle \in E$ is a tuple $\langle \tau^v(v_1), \tau^v(v_2) \rangle$.

In order for the system to be implemented, the vertices must be instantiated with resources and the edges must become active resource dependencies.

For some design $\mathcal{S}$ and an architecture $\mathcal{A}^+$, an *instantiation* of vertices V on a set of resources R is a partial function $q_i : V \nrightarrow R$, such that for any $v \in V, r \in R$ the statement $\langle v, r \rangle \in q_i \Rightarrow \tau^v(v) = \tau^r(r)$ holds true. In other words, resource r can instantiate vertex v if their types match. Relation q_i being a partial function means that for a single instantiation of V each vertex can be instantiated no more than once, however there can be multiple instantiations: $Q = \{q_0, q_1, \ldots\}$. Let $R_i = \text{ran} q_i$ be the set of resources involved in the instantiation q_i. For any pair of instantiations $q_i \in Q, q_j \in Q, i \neq j$ and any resource $r \in R$, it is required that $r \in R_i \Rightarrow r \notin R_j$, i.e. any resource cannot belong to multiple instantiations at the same time.

Let $D \subseteq R_i \times R_i$ be the set of active resource dependencies, w.r.t. the instantiation q_i. D is constrained by E, i.e. for every $\langle r_1, r_2 \rangle \in D$ there must exist $\langle v_1, v_2 \rangle \in E$, such that $q_i(v_1) = r_1$ and $q_i(v_2) = r_2$. Also, following from the properties of vertex instantiation, any edge in E can have at most one related element in D.

Thus, an *implementation* of the design $\mathcal{S}$ on the architecture $\mathcal{A}^+$ is defined as $S_i = \left(\mathcal{S}, \mathcal{A}^+, q_i, D\right)$, where q_i is a vertex instantiation and D is a set of active resource dependencies. The design $\mathcal{S}$ can be mapped onto an architecture giving zero or more implementations (Fig. 2.3):

$$\mathcal{S} \mapsto \mathcal{A}^+ = \{S_0, S_1, \ldots\}. \tag{2.1}$$

Lastly, an implementation is *complete* if $\text{dom} q_i = V$ and for any $\langle v_1, v_2 \rangle \in E$ there exist $\langle q_i(v_1), q_i(v_2) \rangle \in D$, i.e. every vertex and every edge of the system design graph are realised and active. The architecture may have more resources than it is required for the system implementation S_i, but it must have at least the required number of resources and dependencies in order to complete S_i. When it is not possible to form any complete implementations on the architecture $\mathcal{A}^+$ for the system design $\mathcal{S}$, the design is called *non-implementable* on the given architecture. For example, the design is surely non-implementable if it does not share common types with an architecture: $T^v \cap T^r = \emptyset$.

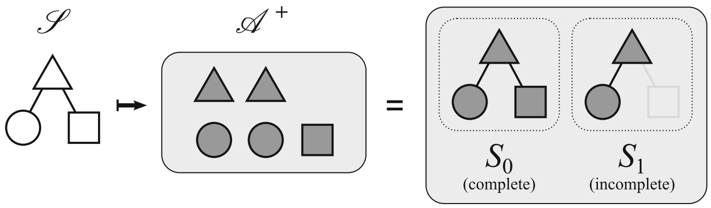


Fig. 2.3 System design $\mathcal{S}$ is mapped onto an architecture $\mathcal{A}^+$ giving system implementations $\{S_0, S_1, \ldots\}$, some or all of which may be incomplete

One can notice that vertex instantiations as well as active dependencies can have multiple solutions for each mapping $\mathcal{S} \mapsto \mathcal{A}^+$. In fact, the result depends on the order in which the elements of q_i and D have been added. That is why we consider this mapping as a non-deterministic discrete process.

2.2.2 Dynamic Systems and Architectures

In many real-life systems the dependencies between resources do not have to be maintained all the time in order for the system to function normally. In fact, for some systems the functionality requires switching dependencies on and off. For example, let's consider a FIFO connection consisting of reader, writer and a buffer. Reader and writer both depend on the buffer, so the system design contains edges $\langle \text{reader}, \text{buffer} \rangle$ and $\langle \text{writer}, \text{buffer} \rangle$. However, at runtime writer and reader accessing the buffer at the same time will cause deadlock, so these dependencies cannot be active at the same time.

In the previous section we mentioned that the mapping $\mathcal{S} \mapsto \mathcal{A}^+$ is a discrete process. Now we can go further by allowing elements of q_i and D to be not just added, but also removed from the implementation. Thus, the state of the dynamic set $q_i \times D$ encodes the state of the implementation S_i. Each state of a dynamic implementation S_i is called a *configuration*. We consider four possible transitions between configurations: instantiation of a vertex, releasing of a resource (as opposed to instantiation), activation of a dependency and deactivation of a dependency:

Instantiate vertex $v \in V$: if v is currently not instantiated in q_i and there is a resource $r \in R$ of the same type as the vertex v and this resource is not used in any other instantiations, add $\langle v, r \rangle$ to q_i.

Release vertex $v \in V$: if v is instantiated in q_i and resource $q_i(v)$ is not used in any active dependency, then remove v from q_i.

Activate dependency $\langle v_1, v_2 \rangle \in E$: if vertices v_1 and v_2 are instantiated in q_i but dependency $\langle q_i(v_1), q_i(v_2) \rangle$ is not active, activate it.

Deactivate dependency $\langle v_1, v_2 \rangle \in E$: if vertices v_1 and v_2 are instantiated in q_i and dependency $\langle q_i(v_1), q_i(v_2) \rangle$ is active, deactivate it.

On the same limited set of resources it is possible to have more working dynamic systems than static, because the dynamic systems can use resources "in turns" grabbing and then releasing them, e.g. as a CPU core activating computing resources according to the instruction being executed [20]. Sharing in the static systems is not possible.

It is important to note that, although an implementation S_i of the system is dynamic, the design of the system $\mathcal{S}$ remains static by definition and includes all vertices that can be instantiated and all dependencies that can be active.

Similarly to dynamic implementations, we can introduce dynamic architectures, where the set of resource R is dynamic, i.e. individual resources of an architecture may appear and disappear during runtime. The closest example is dark silicon [14], which exploits powering on and off different regions of the electronic system. Resources being excluded from R may also model malfunction of these resources. In this case, if some resource used in an active dependency in S_i leaves the architecture, then the system implementation S_i fails.

We illustrate the above definitions by a small example shown in Fig. 2.4. There are three resources A, B and C and two possible resource dependencies $ab = (A, B)$ and $ac = (A, C)$. When the system is fully shut down, no resources and no resource dependencies are required and can therefore be powered off to save energy, as illustrated by the empty box at the top of the diagram. The system can function normally in one of two modes as determined by a Boolean variable *mode*: when *mode* $= 0$ resources A and B as well as the dependency ab must be active, and

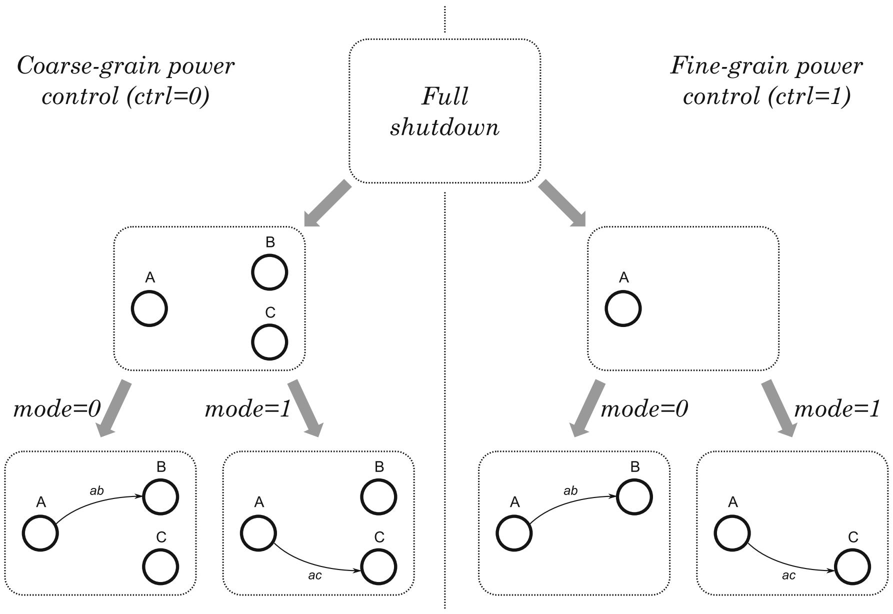

Coarse-grain power control (ctrl=0)
Full shutdown
Fine-grain power control (ctrl=1)
A
B
C
mode=0
mode=1
ab
ac

Fig. 2.4 An example of a dynamic resource graph with completeness condition (2.2)

when *mode* $= 1$ resources A and C as well as the dependency ac must be active. Let us now consider two possible power control strategies. The *coarse-grain control*, activated when $ctrl = 0$, powers on all resources regardless of the current mode, while the *fine-grain control*, activated when $ctrl = 1$, power on the resources on demand according to the current mode. Figure 2.4 covers all possible situations with the four boxes at the bottom of the diagram corresponding to the four possible complete implementations. One can derive the following *completeness condition* which captures all situations in a compact form:

$$A \wedge (\overline{mode} \wedge ab \wedge B \ \vee \ mode \wedge ac \wedge C). \tag{2.2}$$

Note that the condition does not depend on the chosen power control strategy *ctrl*, because it does not influence the completeness property. See [19] for a systematic approach to the derivation of such completeness conditions.

2.2.3 Resources Quantification and Reward Functions

In very large systems, representing each resource with a node leads to large increases in model sizes. An unconstrained architecture, defined in Sect. 2.2.1, differentiates the resources by their types, so it makes sense to group same-type resources under a single node with an added scalar value representing the quantity.

For some resource $r \in R$, its *quantity* $\omega(r) \in \mathbb{Z}^+$ is the number of the resource's instances. For some dependency $\langle r_1, r_2 \rangle \in D$, its *multiplicity* $\omega(\langle r_1, r_2 \rangle) \in \mathbb{Z}^+$ means that the number $\omega(\langle r_1, r_2 \rangle)$ of resource instances r_2 is dependent on the same number of resource instances r_1; $\omega(\langle r_1, r_2 \rangle) \leq \omega(r_1)$ and $\omega(\langle r_1, r_2 \rangle) \leq \omega(r_2)$.

Figure 2.5 gives an example of using resource quantification. The system design shown in Fig. 2.5a represents a task t running on a core c using some scheduler $s = \langle q_i(t), q_i(c) \rangle$. We map this design on the architecture shown in Fig. 2.5b, which consists of n cores $c_0, \ldots, c_{n-1}$ and a number of task instances. Task instances are

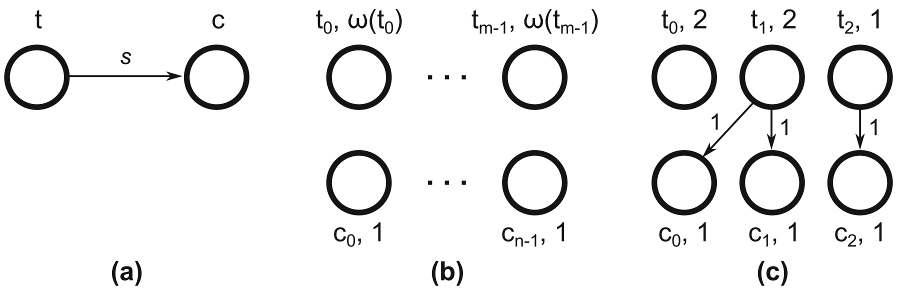


Fig. 2.5 Resource quantification example: (**a**) the system design representing a task t scheduled on a core c, (**b**) the system architecture consisting of n cores and m types of tasks, (**c**) a possible configuration for $n = 3, m = 3$

grouped by their type $t_0, \ldots, t_{m-1}$, m types in total; $\omega\left(t_j\right) \in \mathbb{Z}^+$ is the number of task instances of a particular type t_j, $0 \leq j < m$. Figure 2.5c shows a possible configuration for $n = 3, m = 3$. The system has two tasks of type 1 ($\omega\left(t_1\right) = 2$), thus the node t_1 can have two outgoing connections. Each core, in turn, has a quantifier of 1 and can connect to no more than one task, thus t_0 cannot be scheduled due to the lack of core resources.

In addition to quantifying discrete resources, we can model continuous resources by quantising them in a way that some amount of a continuous resource is represented with an integer ω. A dynamic architecture can also include an inflow and outflow of resource quanta.

An alternative approach, inspired by Markov analysis [17], is using reward functions. For a given configuration of a resource graph, the reward function computes some numeric value representing an instantaneous quantity of a continuous resource. For example, this could be an instantaneous power consumption in a system, and a time-integral of this reward would give the total amount of energy consumed. Notably, in many cases, reward functions can be computed locally on dependency arcs and adjacent resources.

2.2.4 Constrained Architectures

In general, an architecture is defined as $\mathcal{A} = (R, L, T^r, \tau^r)$, where R is the set of discrete resources, T^r is the set of resource types, $\tau^r : R \to T^r$ is the resource type assignment function. The *constraint* $L \subseteq R \times R$ is the set of allowed resource dependencies, so for any pair of resources $r_1 \in R, r_2 \in R$ the statement $\langle r_1, r_2 \rangle \in D \Rightarrow \langle r_1, r_2 \rangle \in L$ must be true, where D is a set of active dependencies of some system implementation $S_i = (\mathcal{S}, \mathcal{A}, q_i, D)$. Thus, an architecture is also a resource graph, and an unconstrained architecture $\mathcal{A}^+$ is a complete graph: $L = R \times R$. The process of mapping the design onto an architecture, $\mathcal{S} \mapsto \mathcal{A}$, is the process of mapping one graph onto another graph.

An illustrative example for constrained architectures is many-core processors. Every core in such a system contains a similar set of resources, like registers and ALUs, but the resources of one core cannot directly access the resources of another core. Instead, we need to connect them via the network component of the system. This can be modelled using architecture constraint.

To conclude Sect. 2.2, the presented resource-driven modelling approach allows the capturing of static and dynamic knowledge of a system being implemented on a given architecture. Both the system design and the architecture are represented with resource graphs, and the behaviour is realised in a transitional semantic. Quantitative modelling of the system resources is also possible using quantised resources or by defining reward functions.

By the definition of resource graphs, anything can be considered a resource. Can we say that the edges of a graph are also resources? It is actually true, especially for hierarchical systems and architectures. This contradiction is explained and solved by Order Graphs in the next section.

2.3 Hierarchical Modelling in Order Graphs

The following discussion revisits the definition of a hierarchy as a sequence of model transformations, which thereafter is applied to graph models leading to Order Graphs. The latter combines the notions of resource modelling with the hierarchical representation of system layers.

2.3.1 Introducing Hierarchies

An underlying approach for having adjustable fidelity in the models relies on different levels of abstraction. Naturally, these layers have to be consistent with each other, however the very definition of consistency may vary from model to model and depend on the system properties that need to be preserved.

A common way to define a model of a system is to represent it as a set tuple $M = (E_1, E_2, \dots E_n)$, where each set E_k contains system elements of a particular type, e.g. vertices, edges, labels, etc. We can also generalise these to a single type—"system elements", $\mathcal{E}$—so $E_1 \subset \mathcal{E}, E_2 \subset \mathcal{E}, \dots$. Thus, we can have a type-agnostic representation of a model: $\mathcal{M} = E_0 \cup E_1 \cup \dots \cup E_n$.

Let M_a and M_b be some system models with corresponding sets of system elements $\mathcal{M}_a, \mathcal{M}_b$, and some relation between these elements $\gamma \subseteq \mathcal{M}_a \times \mathcal{M}_b$. Given a boolean predicate Φ, such that

$$\Phi : \mathbb{P}\left(M_a\right) \times \mathbb{P}\left(M_b\right) \times \mathbb{P}\left(\mathcal{M}_a \times \mathcal{M}_b\right) \to \{0, 1\}, \tag{2.3}$$

the relation γ is called a *consistency relation* between models M_a and M_b under the predicate Φ if $\Phi\left(M_a, M_b, \gamma\right) = 1$. Φ is called the *rule set*, and for convenience can be specified as a conjunction of smaller predicates of the same type (2.3).

The predicate Φ is called *strongly consistent* if it requires γ to be a total surjective relation, i.e. for every element in $\mathcal{M}_a$ there must be at least one related element in $\mathcal{M}_b$, and for every element in $\mathcal{M}_b$ there must be at least one related element in $\mathcal{M}_a$. In this case, γ is called a *transformation*; transformations are further denoted as $\gamma = \mathcal{M}_a \vdash \mathcal{M}_b$ (or $\gamma = M_a \vdash M_b$ since $\mathcal{M}_a, \mathcal{M}_b$ are derived from M_a and M_b).

Let $\left\{\dots, M^{(k-1)}, M^{(k)}, M^{(k+1)}, \dots\right\}$ be an infinite or finite set of models of the same system, where each $M^{(k)}$ models the system in a specific level of details. An *abstraction hierarchy* is a total order of models where any two adjacent models

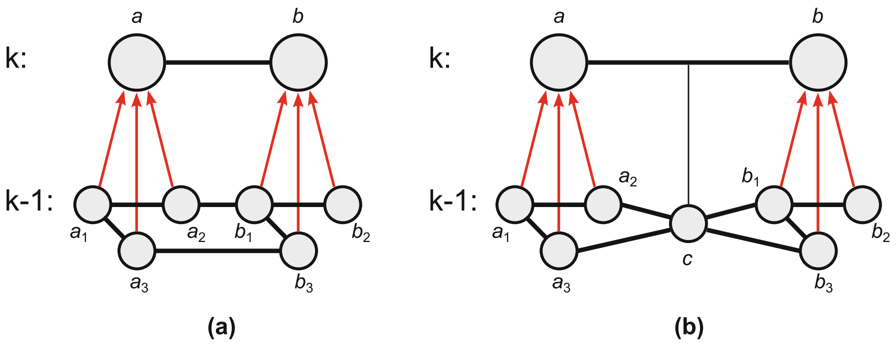


Fig. 2.6 Conventional hierarchy representation (**a**) compared to Order Graphs (**b**); k is the higher level of abstraction and $k-1$ is the lower level

form a transformation $\gamma_k = M^{(k)} \vdash M^{(k+1)}$ under a given strongly consistent predicate Φ_k, and the size of models monotonically decreases (or increases) with k:

$$\mathcal{H} = \cdots \vdash M^{(k-1)} \vdash M^{(k)} \vdash M^{(k+1)} \vdash \cdots \tag{2.4}$$

Each $M^{(k)}$ is kth level of abstraction, also called *order k*.

A hierarchy is called *homogeneous* if it uses the same rule set Φ for all its consistency relations; this implies that $\mathbb{P}\left(M^{(k)}\right) = \mathbb{P}\left(M^{(k+1)}\right)$ for all k.

Every hierarchy contains both horizontal and vertical knowledge: each abstraction layer $M^{(k)}$ is a horizontal view of the system, while the set of relations $\{\ldots, \gamma_k, \gamma_{k+1}, \ldots\}$ stores the information on how different layers interlink. Notions of horizontality and verticality can be found in [10].

Figure 2.6a shows the conventional approach to hierarchical graphs, which is based on clustering and uses tree structures [15]. Each node of a higher layer zooms into a subgraph in a lower layer. Consequently, an edge between two nodes becomes multiple edges between the corresponding subgraphs. The notation used in the diagram is based on Zoom Structures [10]. A convenient way to display graph hierarchies is zoom views, showing verticality and horizontality with vertical and horizontal arcs, respectively. The following is a redefinition of hierarchical graphs in the terms presented in Sect. 2.3.1.

A *Hierarchical graph* is a homogeneous hierarchy, such that, each kth order is a graph $G^{(k)} = (V, E)$, where V is the set of vertices and $E \subseteq V \times V$ is the set of edges; and all consistency relations in this hierarchy are defined as follows:

Inclusion function represents vertex clustering by relating multiple vertices in the lower order to a single vertex in a higher order.

Supplementary inclusion function ensures that all edges within a cluster are also included, i.e. if two vertices in the lower order relate to the same vertex in the higher order, any edge connecting them is automatically related to the same high-level vertex.

Edge grouping function groups edges connecting vertex clusters: an edge in the lower order connects vertices iff there is an edge in the higher order connecting related high-order vertices.

A more formal definition of these rules can be found in [25]. The inclusion function can be chosen arbitrarily, and from it, the other two uniquely describe the edges in the hierarchical graph.

The most important property of the rule set defined above is that it preserves all paths in the graph during the mapping. In other words, for any vertices $v_1, v_2 \in V$ and related vertices $v_1', v_2' \in V'$, if there exists a path between v_1 and v_2 in $G^{(k)}$, there will be a path between v_1' and v_2' in $G^{(k+1)}$, and vice versa:

$$\forall v_1, v_2 \in V, v_1', v_2' \in V' : \gamma_v(v_1) = v_1' \wedge \gamma_v(v_2) = v_2' \Rightarrow \left(P(v_1, v_2) \Leftrightarrow P\left(v_1', v_2'\right)\right), \tag{2.5}$$

where $P(x, y)$ is a function that is true iff there is a path between x and y. This property ensures that the dependencies between resources are consistent throughout the hierarchy.

2.3.2 Order Graphs

Section 2.2 suggests that an edge in a resource graph can be a resource (a node). As an example, let's imagine that Fig. 2.6a models a network interaction, where a is a server and b is a client. On the very abstract level we do not care about the structure of the network, we just need to know that the client and the server are connected somehow, thus we model this entire system as the client and the server connected directly with a single dependency. However, in a more detailed model we can no longer ignore the network protocols and have to introduce it at least as a single resource node as shown in Fig. 2.6b.

A distinct property of the proposed Order Graph (OG) modelling method is that a high-order edge relates to a node at the next lower order. In this case we say that the node *supports* an edge, while in fact this is the same entity viewed from the different abstraction levels. In real-life systems, any dependency is always supported by a resource of some kind, and this "fractal" structure goes down to the smallest details, including atoms and below. We may not want to include all these in the model, and this is pragmatically solved by saying that an edge is either supported by a resource at the lower layer or stays an edge like in conventional hierarchical graphs.

An *Order Graph* is a homogeneous hierarchy, such that, each kth order is a graph $G^{(k)} = (V, E)$, where V is the set of vertices and $E \subseteq V \times V$ is the set of edges; and all consistency relations in this hierarchy are defined as follows:

Inclusion, **supplementary inclusion**, and **edge grouping** are defined as in Sect. 2.3.1.

Support function is a one-to-one mapping of some vertices onto some of the edges of a higher order graph. The first rule on this function tells that we can map a vertex in the lower order to some edge $\langle v_1, v_2 \rangle$ in the higher order iff this vertex is connected to at least one vertex related to v_1 and at least one vertex related to v_2. In addition, all vertices adjacent to v must be related either to v_1 or v_2. Finally, the same vertex cannot be used in a vertex-to-vertex and a vertex-to-edge relation; and the same higher order edge cannot be used in an edge-to-edge and a vertex-to-edge relation.

Supplementary support function groups all edges adjacent to v into the same higher order edge.

These rules are formally defined in [25]. OGs preserve paths in the same way as (2.5) shows for hierarchical graphs.

2.3.3 *Cross-Layer Cuts*

In the approach presented in this chapter, the analysis of the system is performed on a flat model, not the entire hierarchy. The actual benefit of using hierarchies in this case is in the possibility to obtain a flat model (or models) by cutting the hierarchy not horizontally but across multiple layers. The level of details is selected per element of the system, which gives high control on adjusting the precision of the obtained models, ultimately leading to the best sized models for the given fidelity requirement.

An *elementary transformation* is the minimum set of changes that may happen between two graphs without violating the rule set of OGs. Thus, OGs have the following types of elementary transformations, shown in Fig. 2.7:

Inclusion: Vertices and edges of the lower order are mapped into a single vertex in the higher order. Figure 2.7a shows vertices a_1, a_2, a_3, and edge e_1 being mapped into vertex a; relation $\langle e_1, a \rangle$ is implied and not drawn. This elementary transformation also appears in conventional hierarchical graphs.

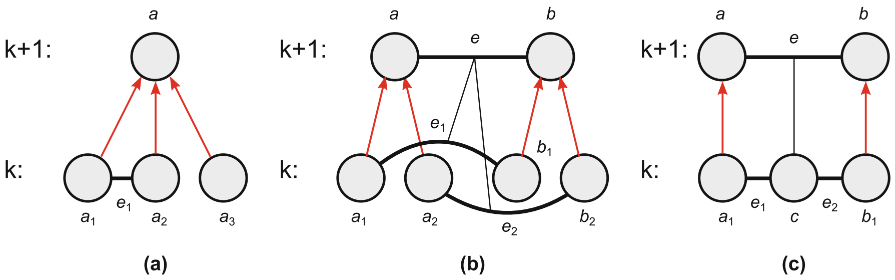


Fig. 2.7 Elementary transformations in Order Graphs and their notation: (**a**) inclusion, (**b**) edge grouping, (**c**) support

Edge grouping: Edges of the lower order are mapped into a single edge in the higher order. Figure 2.7b shows edges e_1, e_2 being mapped into edge e. The relations are drawn as thin black lines to be differentiated from vertex-to-vertex relations. This elementary transformation also appears in conventional hierarchical graphs.

Support: One vertex is mapped into one edge in the higher order. Figure 2.7c shows vertex c being mapped into edge e; relations $\langle e_1, e\rangle, \langle e_2, e\rangle$ are implied and not drawn. This elementary transformation is unique to OGs.

Any transformation $\gamma = G^{(k)} \vdash G^{(k+1)}$ in OG can be represented with a sequence of elementary transformations $\gamma = \gamma_1 \circ \cdots \circ \gamma_n$, or:

$$G^{(k)} \vdash G^{(x_1)} \vdash \cdots \vdash G^{(x_n)} \vdash G^{(k+1)}. \tag{2.6}$$

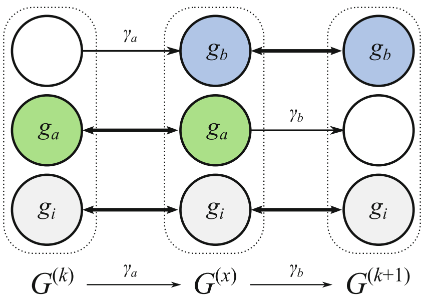
For two consecutive orders $G^{(k)}, G^{(k+1)}$ of an OG, a *cross-layer cut* $G^{(x)}$ between order k and order $(k+1)$ is a graph, such that $G^{(k)} \vdash G^{(x)} \vdash G^{(k+1)}$ under the same rule set, and $G^{(x)}$ is partially equal to $G^{(k)}$ and $G^{(k+1)}$.

Figure 2.8 explains the above definition. Let $\gamma_a = G^{(k)} \vdash G^{(x)}$ and $\gamma_b = G^{(x)} \vdash G^{(k+1)}$. From $\gamma = G^{(k)} \vdash G^{(k+1)}$ and $G^{(k)} \vdash G^{(x)} \vdash G^{(k+1)}$, it follows that $\gamma = \gamma_a \circ \gamma_b$. Then, $G^{(x)}$ can be split in three parts: $G^{(x)} = g_a \cup g_b \cup g_i$, where g_i is the part that is not changed by γ, so $g_i \subseteq G^{(k)}, g_i \subseteq G^{(k+1)}$; $g_a \subseteq G^{(k)}$ is the part not changed by γ_a, and $g_b \subseteq G^{(k+1)}$ is the part not changed by γ_b. Thus, $G^{(x)}$ contains parts equal to subgraphs of $G^{(k)}$ (namely, g_a and g_i) and subgraphs of $G^{(k+1)}$ (g_b and g_i).

An example of a cross-layer cut can be found in Fig. 2.9.

Making a cut through more than two layers—from $G^{(k)}$ to some $G^{(k+b)}$—can be done iteratively. Firstly, obtain a cut between $G^{(k)}, G^{(k+1)}$, so $G^{(k)} \vdash G^{(x_1)} \vdash G^{(k+1)}$. Then, obtain a cut $G^{(x_2)}$ between newly created $G^{(x_1)}$ and $G^{(k+2)}$, which may now contain parts from $G^{(k)}, G^{(k+1)}$ and $G^{(k+2)}$. Repeat the process until the final cut $G^{(x_{b-1})} \vdash G^{(x_b)} \vdash G^{(k+b)}$ is found.

Cross-layer cuts are models of the same system and are consistent with the layers in the corresponding OG and preserve the connectivity property. The choice, which cut is the most appropriate, depends on the application. Section 2.4.3 presents the use case of parametric-proportional approach to optimise the model size for modelling system power.



γ_a
g_b
g_b
g_a
g_a
γ_b
g_i
g_i
g_i
$G^{(k)} \xrightarrow{\gamma_a} G^{(x)} \xrightarrow{\gamma_b} G^{(k+1)}$

Fig. 2.8 Cross-layer cut $G^{(x)}$ explained

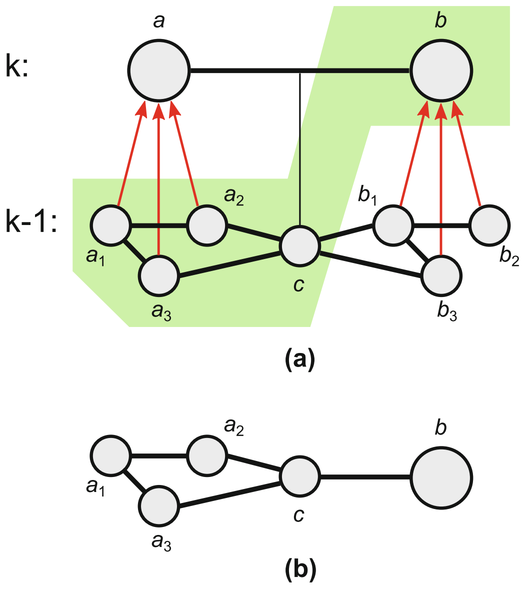


Fig. 2.9 Cross-layer cut example from Fig. 2.6b showing (**a**) the cut and (**b**) resulting flat graph

2.4 Case Studies

This section describes cases of application for our modelling methods. These cases focus on example parameters that can be regarded as resources and we motivate their studies with real applications.

2.4.1 Studying the Performance, Energy and Reliability Trade-Offs of Scalable Systems

In digital VLSI systems, a higher supply voltage (V_{dd}) usually allows a higher operating (clock) frequency and hence a higher throughput, but at the cost of higher power consumption. For reliable operation, there is a limited space within the V_{dd} vs frequency space in which the system can operate, as shown in Fig. 2.10.

Figure 2.10 illustrates the number of limits within which a system can reliably function. There may be a minimum requirement for throughput, which translates to a lower frequency bound a system must achieve. Otherwise the system is not regarded as reliable because it cannot deliver the required performance. There is usually a data reliability requirement which means that the system must operate at a V_{dd} higher than some lower bound because reducing the V_{dd} further the data representation stops being reliably digital and start having unacceptable susceptibility to noise. The VLSI (usually CMOS) technology's fundamental characteristic between its switching speed and V_{dd} means that a circuit cannot be run at a frequency higher

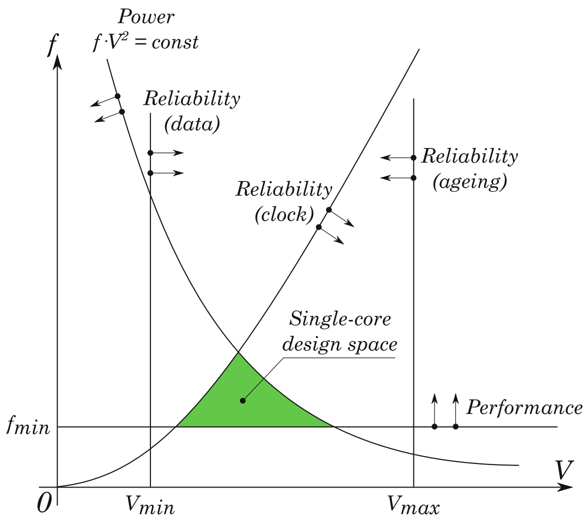


Fig. 2.10 The region of reliable operation

than an upper bound at a particular V_{dd}, in order not to risk combinational logic not completing before the next clock pulse. Any CMOS technology will have a specific highest possible V_{dd} under which it functions correctly. And finally, system power may be limited by some higher bound because of issues like battery life, thermal dissipation, energy efficiency requirements, etc.

Dynamic power is known to be related to switching activity (and through which to system frequency), switching swing voltage (and through which to system V_{dd}) and switching element capacitance (and through which to system size/area—which is a constant before hardware scaling). Lumping all the constants together we can say that power is related to frequency and V_{dd} in the following manner:

$$P = cF \cdot V_{dd}^2, \tag{2.7}$$

where c is a constant, P is the power and F is the frequency. In this section, we explore the issue of core scaling (increasing or decreasing the degree of parallelisation) with an assumption of perfect scaling. Multiplied hardware operating at the same frequency will provide multiplied throughput and require a multiplied amount of power based on the same constant multiplier. Non-zero scaling overheads will be investigated in Sect. 2.4.2. In perfect scaling with a scaling factor of n, the constant c is scaled in the same way, i.e.

$$c = nc_1, \tag{2.8}$$

where c_1 is the c for the hardware before scaling (e.g. a single core). In general, scaling with a factor of n will give a new c which is a factor of n of the unscaled c.

For each core in a new scaled set-up, the available power is also changed by a factor of $1/n$. Considering these factors, for each core, Eq. (2.7) now becomes

$$P_n = cF \cdot V_{dd}^2/n, \tag{2.9}$$

and the overall system power equation with n cores stays the same as (2.7).

Parallelism may be used as a way of increasing throughput without increasing system power dissipation, effectively enlarging the reliable operation region of a system. For instance, if a software application or set of applications can be parallelised and distributed to multiple cores, it will be possible to scale the V_{dd} and the frequency of each core down, while using the multiple cores to improve the overall throughput. Because the dynamic power of CMOS systems is related to V_{dd} and frequency according to Eq. (2.9), reducing both V_{dd} and frequency together reduces power much faster than frequency is reduced, leading to the power-performance advantage of using multiple cores.

Figure 2.11 shows the potential of using parallelisation scaling to reduce power and/or improving performance. The constant power and safe frequency vs V_{dd} operation curves are obtained from experimenting with a real CMOS system [4]. From Fig. 2.11 it can be observed that with a parallelisation scaling factor of 16 and for the constant power budget $P_{\max}$, it is possible to operate at a lower V_{dd} of 0.55 V instead of the nominal 1.2 V, and achieve a nearly four-times overall throughput improvement. On the other hand, if the requirement of throughput is unchanged, it is possible to scale the parallelisation up by the factor of 4, reduce the V_{dd} down to 0.55 V, and use only 25 % of $P_{\max}$ of power.

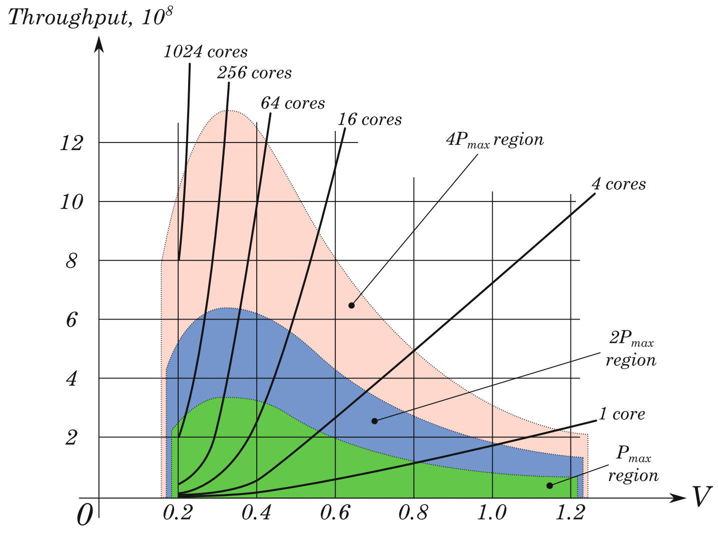


Fig. 2.11 SRAM constant max power curve and scaling lines

Reasoning about these trade-offs can be done in an ad-hoc manner on a per-system or per-circuit bases using characterisation data. However, for system designers it would be much more convenient if this can be done through system-independent models. Resource-driven modelling provides such opportunities as explained in the following sections.

2.4.2 Exploring Concurrency in Many-Core Systems

This section presents the example of using flat dynamic resource graphs, presented in Sect. 2.2, to implement a scalable hardware–software co-simulation for exploring concurrency. The analysis of physical parameters is done via resource access counting. The simulator is flexible towards the hardware architecture and facilitates controllable model fidelity by combining resources from different levels of abstraction. The systematic way of determining the choice of the abstraction levels based on fidelity requirements will be presented in Sect. 2.4.3.

2.4.2.1 Architecture-Open (ArchOn) Simulator

The method to supply resource graphs to the simulation software has been derived from our hardware vision. We view the simulator modules as connected via the connectivity fabric, and the simulator input parser works as a router. Table 2.1 shows some commands for this "router", which provide step-by-step graph configurations as well as explicit invocations of resource state transitions. Applying this method to sparsely connected graphs with many vertices gives more compact specifications than traditionally used adjacency matrices. We call it the *graph assembly language*.

Figure 2.12 shows a virtual communication-based hardware architecture that could potentially emulate most real-world systems. This type of architecture is called transport-triggered architecture [7]. It hasn't become popular in general purpose microprocessors, but it appeared attractive for our purposes. Assuming

Table 2.1 Some commands of graph assembly language

Command	Description
$U[\mathrm{a}] = value$	Set resource *a* state to *value*
$\mathrm{a} \rightarrow \mathrm{b}$	Set a dependency between resources *a* and *b*
$\mathrm{a} \xrightarrow{x} \mathrm{b}$	Set a labelled dependency between resources
$\mathrm{a} \nrightarrow \mathrm{b}$	Unset a dependency
$G = \emptyset$	Clear all dependencies
go!	"Execute" graph: fire all resource state transitions
go to X	Continue assembly from label X (jump)
if condition **go to** X	Conditional jump

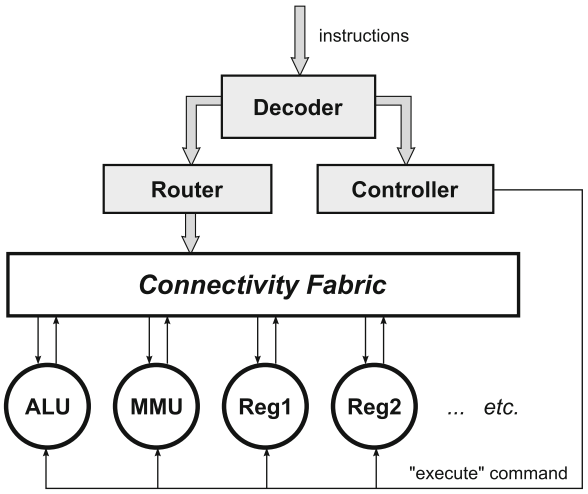


Fig. 2.12 A virtual hardware resembling dynamic resource graphs

that the target system has an instruction set, its software can be recompiled into the connectivity fabric routing commands. The process of executing such software would have alternating phases of configuring the connectivity fabric and executing modules.

2.4.2.2 Benchmark Results

In this section we demonstrate the application of the ArchOn framework to modelling PER trade-offs in a many-core processor. As our example we take the basic computational step in the 3 × 3 matrix convolution that is used in most image processing applications [23]. Given two 3×3 matrices A and B the goal is to multiply them element-wise and sum up the results, denoted by $A \boxtimes B$:

$$\begin{pmatrix} x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 \\ x_7 & x_8 & x_9 \end{pmatrix} \boxtimes \begin{pmatrix} y_1 & y_2 & y_3 \\ y_4 & y_5 & y_6 \\ y_7 & y_8 & y_9 \end{pmatrix} = \sum_{1 \le k \le 9} x_k y_k. \tag{2.10}$$

Usually, one of the matrices is a 3 × 3 sub-matrix of an image being processed and the other matrix, called a *kernel* or a *mask*, represents the required image transformation, e.g., sharpening or edge detection. This step is applied to all 3 × 3 sub-matrices of a given image, each time producing a value for a pixel in the resulting image. This is an embarrassingly parallelisable computation task: one can cut an image into pieces and process them in parallel on different cores. However, the memory access is still a bottleneck, which will define the system's scalability.

As a many-core test platform we used a simplified ARM architecture. The mainstream ARM processors to date consist of up to 8 cores, however there are concrete plans to increase the number of cores to 16, 32, and beyond [2, 12]. In ArchOn we ran the benchmark on up to 64 cores, however it was possible to use even more as the simulation time scales linearly with the number of cores. The difference between multi-core and single core architectures for the simulator is in the restrictions on certain connections between the resources belonging to different cores, as described in Sect. 2.2.4.

Algorithm 1 ARM instruction `MLA r8,r9,r10,r8` in graph assembly language

```
  $G = \emptyset$
  $\text{r9} \xrightarrow{n} \text{mul}$
  $\text{r10} \xrightarrow{m} \text{mul}$
*go!*
  $G = \emptyset$
  $\text{mul} \xrightarrow{n} \text{alu\_add}$
  $\text{r8} \xrightarrow{m} \text{alu\_add}$
*go!*
  $G = \emptyset$
  $\text{alu\_add} \rightarrow \text{r8}$
*go!*
```

The convolution filter software is written in ARM assembly language. Here, a 256×256 image is divided between processing cores, each working on a separate set of pixels (with single pixel wide overlaps). Each pixel is a 32-bit integer representing grey-scale colour. For every ARM instruction we derive a resource evolution and translate it into graph assembly language. This is a routine task since all instructions follow a common pattern. The process of translation can be done automatically. An example instruction is shown in Algorithm 1.

The nature of this simulation requires appropriate functional behaviour resources, thus the computation resources have to be modelled down to the ALU level. On the other hand, the memory access does not require exact modelling of all levels of cache (the behaviour of cache is typically non-deterministic), and can be approximated to the set number of access “modes”. Since shared memory would become the bottleneck while scaling to many-cores, we added control over the “criticality” of this resource, so the program can be executed in three different modes: (1) simultaneous read and write access to the memory is allowed, (2) simultaneous read is allowed, but only one writer is allowed at a time, (3) all memory access is exclusive and must be done sequentially. With this example we start exploring non-ideal concurrency scaling and non-zero overheads.

By Amdahl’s law [1], the theoretical speed-up that can be achieved by executing a given algorithm on a system capable of executing n threads is

$$\frac{T(1)}{T(n)} = \frac{1}{s_s + \frac{s_p}{n}}, \tag{2.11}$$

Table 2.2 Execution time (in cycles) versus the number of cores running for different memory access models

N cores	Mode 1: multiple read multiple write	Mode 2: multiple read single write	Mode 3: single read single write
1	26607635	26607635	26607635
2	13303827	13303947	**19660819**
3	8938515	8938755	**19660819**
4	6651923	**7864625**	**19660819**
5	5404691	**7864625**	**19660819**
6	4469267	**7864625**	**19660819**

where $T(n)$ is the time an algorithm takes to finish when being executed on n cores, and $s_s \in [0, 1]$ is the fraction of the algorithm that is strictly serial, $s_p = 1 - s_s$ is the fraction of the algorithm that runs in parallel.

For our algorithm s_s and s_p are not known in advance, and actually depend on the memory mode and the number of cores running, i.e. are not constants. ArchOn time estimation enables analysis of this factor. Table 2.2 gives the estimates for execution time. From (2.11) we can find s_p, which will be an estimate of parallelisation for our example. In Mode 1 the scaling is nearly perfect, $s_p \approx 9.999999$, however in Mode 3 the memory becomes such a narrow bottleneck that there is no performance gain for more than two cores (performance cap is shown in bold). The most illustrative example is Mode 2, when multiple cores are allowed to simultaneously read, but forbidden to simultaneously write to the memory. The performance cap is reached at four cores, and s_p varies from 9.96 at two cores to 0.4 at four cores and decreasing.

The main goal of this section is to use ArchOn simulation to draw PER diagrams, described in Sect. 2.4.1. The diagrams can be drawn using the ARM power models, which, however, are typically unavailable. ARM power characterisation from measurements using a prototyping board requires high effort, as shown in Sect. 2.4.3.1. Moreover, the commercially available systems usually do not allow near-threshold and sub-threshold operation. In order to explore PER into these regions, we substitute the power profile with that of an asynchronous SRAM controller [4], which qualitatively reflects the behavior of any CMOS combinational logic in terms of PER. Thus, a hypothetical proprietary ARM core with a wide voltage range should display similar PER relations.

Figure 2.13a illustrates perfect scaling (memory Mode 1) with applied power limit of 2 mW. Figure 2.13b shows the PER diagram for the same system with the same power budget after applying actual metrics for scalability to many cores in Mode 2. The diagram considers only integer numbers of cores, hence the performance line looks jagged. Please note that the line for four cores in Fig. 2.13b is lower than in Fig. 2.13a due to imperfect scaling. The data for this graph is computed automatically. One can see that the performance cap is clearly reflected in the power limit.

Such diagrams can be used in a runtime management system in order to predict the best voltage and the number of cores for a particular software with regard to the

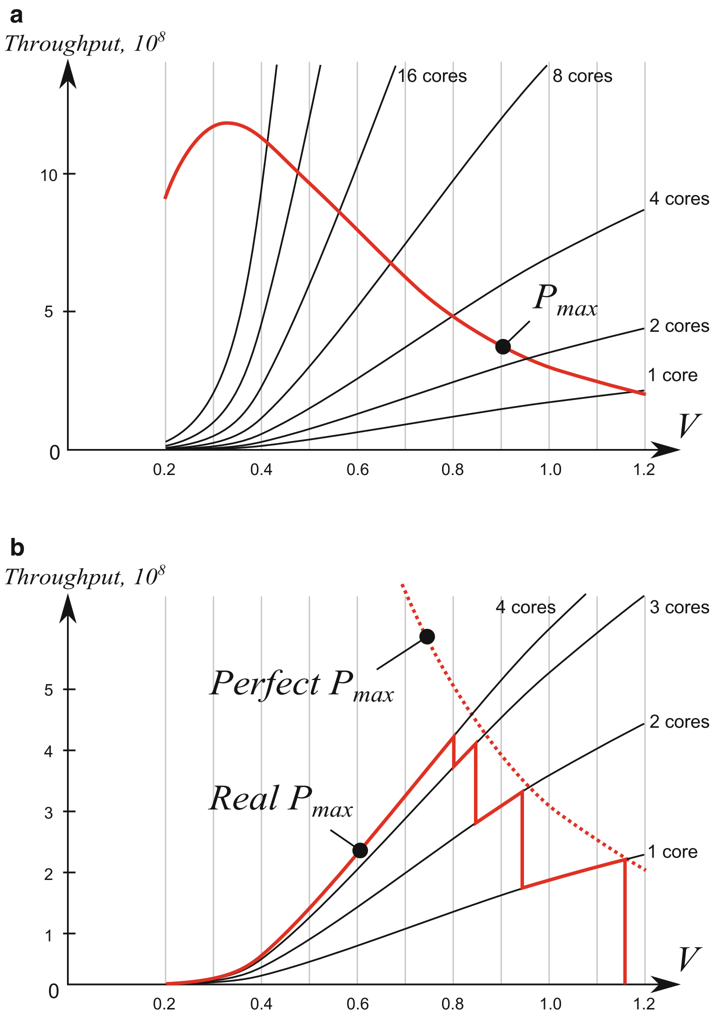


Fig. 2.13 Computed power limit for perfect scaling (**a**) and for an actual scaling to many cores in the simulated example (**b**)

power restrictions. In our example, for memory Mode 2, if the system is limited to 2 mW, the best number of cores is 4 running at 0.8 V.

2.4.3 Power-Proportional Modelling of Heterogeneous Systems

This section presents the method of managing fidelity based on power-proportionality metric and cross-layer cuts. Section 2.3 gives the theoretical foundation for this method.

The method can be applied to resource graph simulations in ArchOn. However, in contrast to ArchOn's deterministic simulations, the example method presented in this section uses stochastic modelling and Markov rewards for quantitative analysis of system power consumption. The reward functions are still determined by the resource graphs, as described in Sect. 2.2.3. Since state-space exploration-based analysis has exponential complexity, choosing the right size of the model becomes crucial.

2.4.3.1 Platform Description

The experimental platform used in this section is Odroid XU3 [22].

The main component of Odroid XU3 is the 28 nm 8-core Application Processor Exynos 5422. This System-on-Chip is based on the ARM big.LITTLE architecture [13] and consists of a high performance Cortex-A15 quad-core processor block, a low power Cortex-A7 quad-core block, a Mali-T628 GPU and 2GB LPDDR3 DRAM. The board contains four real-time current sensors allowing the measurement of power consumption on the four separate power domains: A7, A15, GPU and DRAM. Because of the system's heterogeneity and suppled power measurement facilities, the Odroid XU3 is arguably one of the best off-the-shelf heterogeneous platforms for power analysis.

For each domain, the supply voltage and clock frequency can be tuned through a number of pre-set pairs of values. The performance-oriented A15 quad core block can scale its frequencies from 200 to 2000 MHz, whilst the low-power A7 block has a frequency range from 200 to 1400 MHz. Core 0 in the A7 domain has an additional speciality of running the OS kernel and drivers, and it cannot be switched off. We avoid using this core for stress tests and benchmarks to reduce the impact from the OS on the measurements.

There are compatible Linux and Android distributions available for Odroid; in our examples, we used Ubuntu 14.04.

In our characterisation experiments, firstly the above parameters were measured without any additional workload, with only the OS running. Then the same parameters were measured for each core with application threads running. We experimented with the typical Linux stress task, i.e. running square root calculations repeatedly, and in addition, other computations including logarithm calculations and the four arithmetic operations. We also covered different levels of workload.

Another important experiment is the measurement of the same parameters with some of the cores in each block disabled. Odroid allows from one to four of the A15 to be disabled and from one to three of the A7 to be disabled.

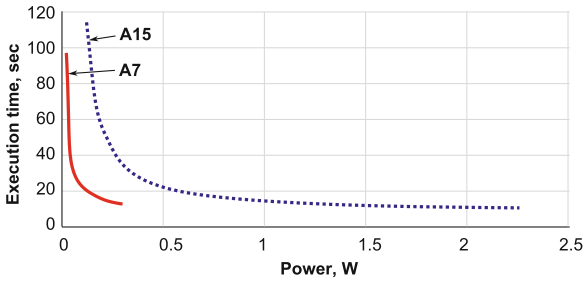


Fig. 2.14 Measured power in relation to the time required to complete a fixed amount of computation. The same performance requires higher power consumption from A15 than A7

In these experiments, it was observed that an A15 consumes four times or more power than an A7 when both are running at the same frequency, and up to an order of magnitude more power when both are running at the same voltage. Figure 2.14 shows the relationship between power consumption and the execution time for the two types of cores on the average running a range of different types of tasks.

These radically different performance and power figures, and their complex relations to the different tasks being executed in a core, validate the approach promoted in this paper. For instance, when certain tasks are mapped to the A7 block, because of the relatively light power demand of these cores we may be able to afford to model such processing with less fidelity, i.e. using a more probabilistic model and/or using a more structurally fuzzy model. For instance, when the A15 block is also running, it may be a good idea to not represent individual A7 cores but to cover the entire A7 block as a meta-core with a single model.

2.4.3.2 Platform Model

In this case study we focus on modelling power consumption of the platform. Two major contributors are task affinities (which task runs on which core) and voltage-frequency pairs.

Figure 2.15 shows the OG model of the system. At the higher levels of abstraction, the system is represented as a set of tasks running on a platform, which in turn consists of a computation component and a power component. The computation resource is provided by A7 and A15 cores, which appear in the lower orders, and the power resource is divided into four power domains, as described in Sect. 2.4.3.1.

For clarity, some of the horizontal edges on this diagram are hidden: every core is actually connected to the corresponding V_{dd} tree and to the task node, etc. Every $\langle\text{Task, Core X}\rangle$ edge at order 0 represents scheduling of a task onto a certain core. The next order groups these edges into $\langle\text{Task, A7 cores}\rangle$ and $\langle\text{Task, A15 cores}\rangle$, respectively. Similarly, every core is connected to the corresponding "V_{dd} tree" resource in the power domain. "V_{dd} tree" resource included in "A15 power domain" supports $\langle\text{A15 cores, A15 power domain}\rangle$ dependency; the same is true for A7

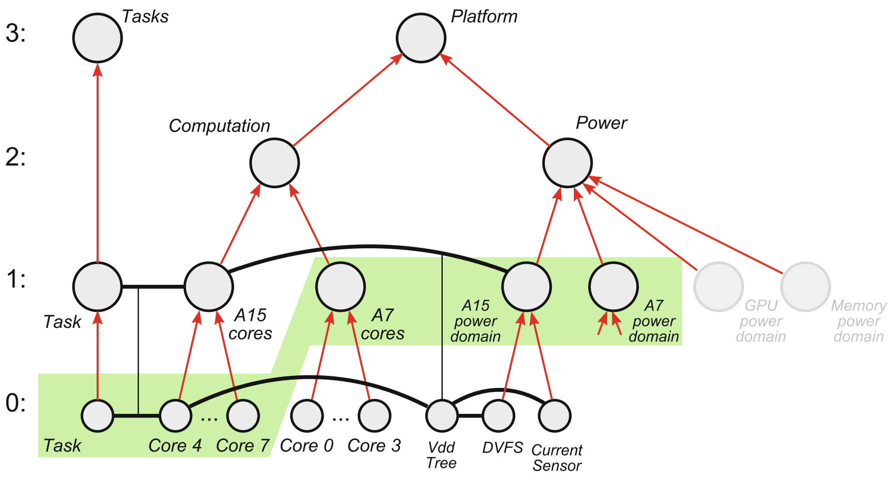


Fig. 2.15 Order Graph model of running tasks on Odriod XU3 platform (some horizontal dependencies are omitted). The *shaded area* marks the cut shown in Fig. 2.17

Cores. In this example, the interactions with "GPU power domain" domain and "Memory power domain" are not captured during the experiments, therefore these resources are removed from the model in the following discussion.

The above OG model represents the structural (static) knowledge of the system. In order to model power, this section uses Stochastic Activity Networks (SANs) [26]. SANs are an extension to Generalised Stochastic Petri Nets (GSPNs) and a more expressive representation language. SANs inherit from Petri nets the basic elements: places, transitions and tokens. In general, a system state is a marking, which intuitively is a configuration of token to place distribution (the number of tokens in each place). A system's dynamic progression consists of its moving from one state to another. This is represented in Petri nets as the firing of a transition. A transition may fire when every one of its input place is marked with at least one token and firing a transition results in a marking modification to a new state, where every input place has its marking reduced by one and every output place has its marking increased by one.

The SANs formalism provides a general way of specifying the enabling of an activity (SANs extension of a Petri net transition), a general way of specifying a completion (firing) rule, a method of representing zero-time events (hence including deterministic as well as stochastic behaviours), a method of representing probabilistic choice in addition to probabilistic delay provided by GSPNs. All of these are based on extensions of the Petri net rules described above. The SANs formalism also provides state-dependent reward values and general delay distributions on activities. For instance, SANs markings can be coded with rewards relating to such parameters as power consumption.

A crucial issue to be modelled for this system, when we talk about system power consumption, is processing, i.e. the execution of threads/tasks in the cores. The fundamental processing element model is shown in Fig. 2.16a. Here the place

Capacity represents the unused capacity of a processing element (e.g. a core), and the place *Processing* represents the current number of tasks being executed in the core. If it is a single core, the sum of tokens of these two places represents the pipeline depth or multi-threading capability of the core. If there are multiple cores in this model, the sum of tokens in the *Capacity* and *Processing* places represents the entire block's multi-threading capability. There is a direct relation between the marking in this model and the resource quantifiers in the graph shown in Fig. 2.5:

$$M\left(Tasks_j\right) = \omega\left(t_j\right) - \sum_i \omega\left(s_{ij}\right),$$

$$M\left(Capacity_i\right) = \omega\left(c_i\right) - \sum_j \omega\left(s_{ij}\right),$$

$$M\left(Processing_{ij}\right) = \omega\left(s_{ij}\right).$$

In the model in Fig. 2.16a, the initial marking, as shown, represents multi-tasking capacity of three and no active processing. The *Spawn* activity is a stochastic activity with a given firing rate and distribution, and because it does not have input places, it is always enabled and can generate tokens into the *Tasks* place. When both *Tasks* and *Capacity* places are marked, the *Start* transition becomes enabled. This is instantaneous transition, thus it fires deterministically and adds a token into *Processing*, whilst removing one token each from *Capacity* and *Tasks*. The input gate of the *Start* transition may contain any additional logic controlling the firing depending on the global of the net. The stochastic transition *Finish* represents the end of processing of a task and returns the token to *Capacity*. The rate of *Finish* represents the throughput related to a single task.

The power consumption is modelled by assigning rewards to markings in appropriate places. For instance, each token in *Processing* can have a reward value corresponding to the power consumption of processing a single task in this core (or cores). A token in *Capacity* can have a reward value associated with the idle power.

Different levels of fidelity are possible with this representation. For instance, the degree of probabilistic vs. deterministic can be tuned for a more or less fuzzy representation. We may decide to model part of a core (i.e. a multiplier), an entire

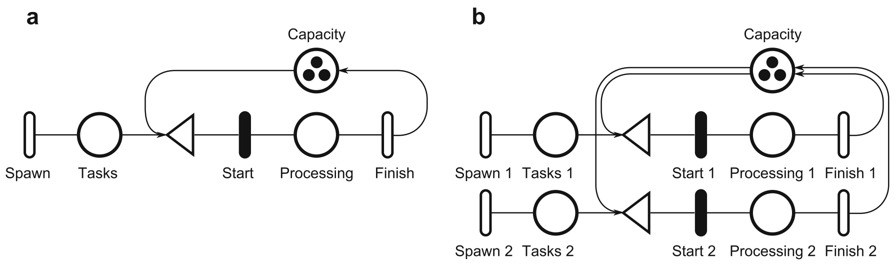


Fig. 2.16 SANs models for task execution

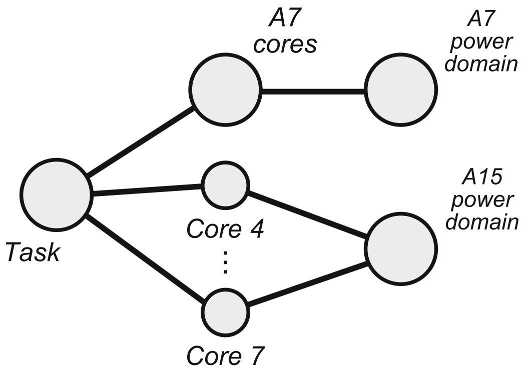


Fig. 2.17 Proposed cross-layer cut for power-proportional modelling

single core, a core-block, or the entire Odroid chip with one of these sub-nets. When setting up a more detailed model with higher fidelity, we may need to distinguish how a processing element behaves with different types of tasks, as our experiments showed that the Odroid cores consume different amounts of power when dealing with different tasks. The model in Fig. 2.16b allows this differentiation by assigning different rewards to *Processing* places corresponding to different tasks. With more fuzzy representations, such issues may be covered by probabilities.

Once a cut has been determined using the OG model, a flat SANs model covering the entire system can be made with different levels of fidelity for different parts. This will be a flat model with power-proportional fidelity and effort.

2.4.3.3 Power-Proportional Model Sizes

Based on experimental data from the Odroid, presented earlier in Sect. 2.4.3.1, for a certain modelling fidelity we may need to represent each A15 core with a model of the type in Fig. 2.16b, with multiple types of tasks—e.g., CPU-heavy and memory-heavy, and many levels of DVFS and workload resolutions. For the same level of fidelity, we can represent the entire A7 block with a single sub-net of the type in Fig. 2.16a without task, DVFS and workload differentiation.

The corresponding OG cut is shown as a flat graph in Fig. 2.17 and also marked with a shaded area in Fig. 2.15. Power-proportional cuts through the model space usually result in models whose sizes are optimal for studying power, in the sense that the resolution or fidelity of power as a parameter is constant through the model. In other words, a power-proportional cut for a specific power representational fidelity gives the smallest possible model for that degree of fidelity. Other representations away from this cut will inevitably result in certain parts showing an unnecessarily higher degree of fidelity leading, usually, to higher degrees of representational complexity.

This approach can be expressed with a metric (p/ec), where p is the power consumption (or any other parameter in study) of a resource, e is the modelling error produced locally by this resource's sub-model and c is the cost to compute this sub-model. The most power-proportional cut has minimal variance of this metric across all its elements. Typically, reducing the error in the model increases its computational cost, hence the term ec within the same modelling technique can be approximated to a constant. This gives the direct proportionality metric p meaning that each resource in the final cross-layer model should have as approximately the same as possible power consumption as the other resources.

One of the generally accepted metrics of model size and therefore modelling effort and the effort of using models is the size of the state space of the model. For instance, one of the envisaged applications of our modelling method is the design and analysis of runtime parameter management algorithms or machines, e.g. runtime power management for mobile and embedded systems. For such management or control schemes, more sophistication is usually needed to achieve better results. Naïve examples that are widely available in the public and commercial domains, such as such Linux/Android power governors as `ondemand`, usually assume very simplistic plant models and rely on feedbacks to achieve some degree of effectiveness, which is almost never optimal. More sophisticated algorithms such as those based on learning and those providing a degree of adaptation can almost always provide better results than the standard governors, but inevitably require better plant models. On the other hand, most computer system control algorithm designers are most comfortable with thinking of the plant as a state machine. And many types of parameter management algorithms, e.g. learning and model adaptive schemes, depend on a state space representation of the plant being available [8, 9, 18, 27]. The size of the state space of a model, therefore, is directly relevant for this type of model usage.

The example architecture of the type seen in the big.LITTLE Exynos chip featured in the Odroid system consists of N power domains. The kth power domain, $0 \leq k < N$, has d_k DVFS points (pre-set pairs of V_{dd} and clock frequency values) and c_k processing cores of the same type. For such a system and for power studies, the fundamental state element is 'a particular core in a particular power domain is running a particular type of task at a certain workload'. Usually the parameter representational fidelity requirement dictates the granularity of workload representation, which should be constant within each individual power domain, as there is no intra-domain core heterogeneity. This leads the jth core in the kth domain having w_{kj} workload points and t_{kj} types of tasks, where $0 \leq j < c_k$. With no core heterogeneity within a power domain, it is convenient to differentiate workloads and task types into the same numbers of points for all cores, i.e. w_k and t_k. The size of the state space S of a cut model for such a system as the controlled plant, of the type described in the previous sections, is therefore

$$|S| = \prod_{k=0}^{N-1} d_k \left(w_k t_k\right)^{c_k} . \tag{2.12}$$

For the Odroid's ARM cores, reducing the representation of the A7 cores to a single execution sub-net model, as in Fig. 2.17, effectively changing c_{A7} from 4 to 1, produces $(w_{A7}t_{A7})^3$ times reduction of the model state space. Reducing the DVFS resolution of the A7 cores to equal power fidelity of the A15 cores produces a further state space reduction.

There are 20 DVFS points for the A15 power domain and 15 for the A7 power domain in the Odroid. Maintaining the same power fidelity, a maximum of five, not 15, DVFS bands are needed for the A7 domain in the model. Assuming a workload resolution of five (0, 25, ..., 100 %) and task type resolution of two (CPU- and memory-heavy), a $3 \times (5 \times 2)^3 = 3000$ times reduction of the state space (new state space size = 1/3000 or 0.033 % of the original) can be obtained by using a power-proportional cut. This kind of state space reduction may result in qualitative differences in the sophistication of the runtime management scheme given any constant overhead budget for the management, or it can be used to reduce the management overhead whilst maintaining the same degree of management effectiveness.

This modelling method provides more opportunities for runtime tuning and adaptation because cuts may be allowed to dynamically change during runtime. For instance, if it is found that no A15 core is active and the entire A15 block is shut down, power fidelity may be improved by representing the A7s individually by adopting a different cut. This can be necessary because an A15 total shutdown may indicate that the system is running in low power or even survival modes and during these modes what are regarded as small amounts of power during normal operation become significant. This should lead to a higher degree of fidelity in representing the quantity of power in the runtime model. On the other hand, if the A15 total shutdown is purely a result of workload demand reduction, the previous coarse A7 block cut should be entirely satisfactory. The facility of layer-crossing cuts provides additional flexibility for model adaptation.

2.5 Conclusions

This chapter presents a general systematic approach to model complexity control through managing model fidelity. The approach is based on resource-driven modelling and includes two concrete methods. Resource graphs represent the static information of computation systems as sets of resources and their cross dependencies, and the dynamic behaviour of these systems as evolution steps of resources and dependencies. This method allows the straightforward emphasis of elements and issues that can be regarded as resources and their interplay in system models, making it easy for designers to reason about them. Issues such as the level of representational fidelity of parameters can be managed through resource definitions leading to the scalability of models as well as the straightforward reasoning of the scalability of systems themselves.

The second method presented within this resource-driven framework is the use of cross-layer cuts to achieve parameter-proportional fidelity in hierarchical models. In this a previously presented formalism Order Graphs is shown to be effective, and techniques relevant to the derivation of parameter-proportional cuts are presented. We propose a metric for rationalising parameter fidelity across complex models within this context.

The entire approach is tested and validated by a number of application cases, with systems ranging from homogeneous many-core to heterogeneous multi-core, and properties ranging from performance, energy/power and reliability. Models are derived for design-time explorations, and both static and dynamic analysis are made. For analysis, the usefulness of such models is demonstrated through both simulation studies and state space analysis.

Acknowledgements This work is supported by EPSRC grant EP/K034448/1.

References

1. G.M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, in *Proceedings of the Spring Joint Computer Conference, AFIPS'67* (Spring) (ACM, New York, 1967), pp. 483–485
2. ARM. http://www.arm.com, 2015
3. G. Balbo, Introduction to generalized stochastic petri nets. *Formal Methods for Performance Evaluation*. Lecture Notes in Computer Science, vol. 4486 (Springer, Berlin, 2007), pp. 83–131
4. A. Baz, D. Shang, F. Xia, A. Yakovlev, Self-timed SRAM for energy harvesting systems. J. Low Power Electron. **7**(2), 274–284 (2011)
5. A. Beyranvand Nejad, A. Molnos, K. Goossens, A unified execution model for multiple computation models of streaming applications on a composable MPSoC. J. Syst. Archit. **59**(10), 1032–1046 (2013)
6. S. Borkar, Thousand core chips: a technology perspective, in *Proceedings of the 44th Annual Design Automation Conference, DAC'07* (ACM, New York, 2007), pp. 746–749
7. H. Corporaal, Design of transport triggered architectures, in *Proceedings on Design Automation of High Performance VLSI Systems* (1994), pp. 130–135
8. A. Das, R.A. Shafik, G.V. Merrett, B.M. Al-Hashimi, A. Kumar, B. Veeravalli, Reinforcement learning-based inter- and intra-application thermal optimization for lifetime improvement of multicore systems, in *Proceedings of the 51st Annual Design Automation Conference*, DAC'14 (ACM, San Francisco, 2014), pp. 1–6
9. A.K. Das, R.A. Shafik, G.V. Merrett, B.M. Hashimi, A. Kumar, B. Veeravalli, Workload uncertainty characterization and adaptive frequency scaling for energy minimization of embedded systems, in *Proceedings of the Conference on DATE'15*, March 2015
10. A. Ehrenfeucht, G. Rozenberg, Zoom structures and reaction systems yield exploration systems. Int. J. Found. Comput. Sci. **25**, 275–306 (2014)
11. H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, D. Burger, Dark silicon and the end of multicore scaling, in *Proceedings of the 38th Annual International Symposium on Computer Architecture, ISCA'11* (ACM, New York, 2011), pp. 365–376
12. EZchip. http://www.tilera.com, 2015
13. P. Greenhalgh, *big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7 – Improving Energy Efficiency in High-Performance Mobile Platforms*. ARM, 2011. White Paper

14. N. Hardavellas, M. Ferdman, B. Falsafi, A. Ailamaki, Toward dark silicon in servers. IEEE Micro **31**(4), 6–15 (2011)
15. B. Kumar, E.S. Davidson, Computer system design using a hierarchical approach to performance evaluation. Commun. ACM **23**(9), 511–521 (1980)
16. Y. Lhuillier, M. Ojail, A. Guerre, J.-M. Philippe, K. Ben Chehida, F. Thabet, C. Andriamisaina, C. Jaber, R. David, HARS: a hardware-assisted runtime software for embedded many-core architectures. ACM Trans. Embed. Comput. Syst. **13**(3s), 102:1–102:25 (2014)
17. Q.-L. Li, Markov reward processes. *Constructive Computation in Stochastic Models with Applications* (Springer, Berlin, 2010), pp. 526–573
18. L.A. Maeda-Nunez, A.K. Das, R.A. Shafik, G.V. Merrett, B. Al-Hashimi, PoGo: an application-specific adaptive energy minimisation approach for embedded systems, in *HiPEAC Workshop on Energy Efficiency with Heterogenous Computing (EEHCO). HiPEAC*, January 2015
19. A. Mokhov, Conditional partial order graphs. PhD thesis, University of Newcastle upon Tyne, School of Electrical, Electronic and Computer Engineering, 2009
20. A. Mokhov, A. Iliasov, D. Sokolov, M. Rykunov, A. Yakovlev, A. Romanovsky, Synthesis of processor instruction sets from high-level ISA specifications. IEEE Trans. Comput. **63**(6), 1552–1566 (2014)
21. A. Nouri, M. Bozga, A. Molnos, A. Legay, S. Bensalem, Building faithful high-level models and performance evaluation of manycore embedded systems, in *In Proceedings of 12th ACM/IEEE International Conference on Methods and Models for System Design, MEMOCODE*, 2014
22. Odroid XU3. http://www.hardkernel.com/main/products, 2013
23. M. Petrou, C, Petrou, *Image Processing: The Fundamentals* (Wiley, Chichester, 2010)
24. A. Rafiev, A. Iliasov, A. Romanovsky, A. Mokhov, F. Xia, A. Yakovlev, Studying the interplay of concurrency, performance, energy and reliability with ArchOn – an architecture-open resource-driven cross-layer modelling framework, in *Proceedings of International Conference on ACSD*, 2014
25. A. Rafiev, F. Xia, A. Iliasov, R. Gensh, A.M.M. Aalsaud, A. Romanovsky, A. Yakovlev, Order graphs and cross-layer parametric significance-driven modelling, in *Proceedings of International Conference on ACSD*, 2015
26. W.H. Sanders, J.F. Meyer, Stochastic activity networks: formal definitions and concepts, in *Lectures on Formal Methods and Performance Analysis* (Springer, Berlin, 2001), pp. 315–343
27. A. Suardi, S. Longo, E.C. Kerrigan, G.A. Constantinides, Robust explicit MPC design under finite precision arithmetic, in *Proceedings of IFAC*, 2014
28. B. Wang, Y. Xu, R. Rosales, R. Hasholzner, M. Glaß, J. Teich, End-to-end power estimation for heterogeneous cellular LTE SoCs in early design phases, in *2014 24th International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, Sept 2014, pp. 1–8

Chapter 3
Empowering Mixed-Criticality System Engineers in the Dark Silicon Era: Towards Power and Temperature Analysis of Heterogeneous MPSoCs at System Level

Kim Grüttner

3.1 Introduction

The vision of ambient intelligence describes an "intelligent environment," which reacts in a sensitive and adaptive way to the presence of people and objects, offering a variety of services. These services will combine data from various sources creating valuable information and support the self-organization across different domains like transportation, medical, and energy. They will be an integral part of smart objects (e.g., consumer electronics, smart phones, cloth, cars, buildings, traffic infrastructure, etc.) and thus of our daily life. Current technologies trend will help enable the vision of ambient intelligence:

– Multi-core processors that provide increasing computation power under acceptable computation/power ratio even for mobile battery-powered systems
– Increasing semiconductor integration levels enable powerful on-chip networking facilities that enable on-chip scaling of computation resources and seamless integration with networked distributed systems
– Increasing heterogeneity (more than Moore) enables sensor and actuator integration in a single chip or system in package

With the support of many others, see Acknowledgments.

K. Grüttner (✉)
OFFIS — Institute for Information Technology, Oldenburg, Germany
e-mail: kim.gruettner@offis.de

A. Molnos, C. Fabre (eds.), *Model-Implementation Fidelity in Cyber Physical System Design*, DOI 10.1007/978-3-319-47307-9_3

– Wireless communication technologies enable flexible interconnect topologies among devices of all kinds.

All of these enabling technologies are driven by the consumer market (mobile phones, tablets, laptops, etc.) and redefine the way embedded and complex systems are being designed. This way, safety- and mission-critical services that have been previously running on dedicated and often custom designed hardware/software platforms will either interact, be tightly connected or even execute on devices made of generic hardware/software platforms that have been previously used with non-safety/mission-critical applications only.

The advantages are clear in terms of reduced Space/Size, Weight, and Power consumption (SWaP). Generic hardware platforms with multiple processing cores and flexible programmable I/O logic have a much lower price than custom designed platforms. Scaling the number of cores per chip instead of controllers and ASICs, combined with the flexibility of pure software solutions, offers the required scalability and flexibility for ambient intelligence. With these technological capabilities, integration of critical and non-critical services on the same hardware platform becomes essential. As observed today, the ongoing convergence of devices, which leads to an increased function/service density (number of functions/services per device), will rapidly break the existing separation of critical and non-critical services running on isolated devices.

With critical and non-critical applications integrated onto a single chip locally, enforcing temporal and spatial segregation becomes an essential capability in the design of multi-core platforms, operating systems, and applications rather than segregating them through allocation and mapping onto dedicated, physically separate devices. When moving to smaller technology nodes, segregation techniques must cope with totally new challenges (see Fig. 3.1). While extra-functional properties like cost, installation space, and weight can be optimized, by taking advantage of higher integration density, the main challenge is to guarantee temporal and spatial segregation on a single chip, while maintaining power, temperature, and reliability requirements. In contrast to the traditional distributed multi-controller approach, power, temperature, and reliability cannot be considered as isolated effects (per device) for single chip mixed-criticality systems.

Significant improvements have been achieved to support the design of mixed-critical systems by developing predictable computing platforms and mechanisms for temporal and spatial segregation between applications of different criticalities sharing the same computing platform [2]. Such platforms provide techniques for the compositional certification of applications' correctness, run-time properties, and reliability of computing platform. Multi-core SoCs that supports spatial and temporal segregation for mixed-critical applications need to become analyzable regarding their power, temperature, and reliability properties taking into account a certain configuration and application mapping [7].

The reminder of this chapter is organized as follows. In the next section an introduction to the CONTREX project is given. Section 3.3 describes the CONTREX flow, considering power and temperature of a full chip in virtual

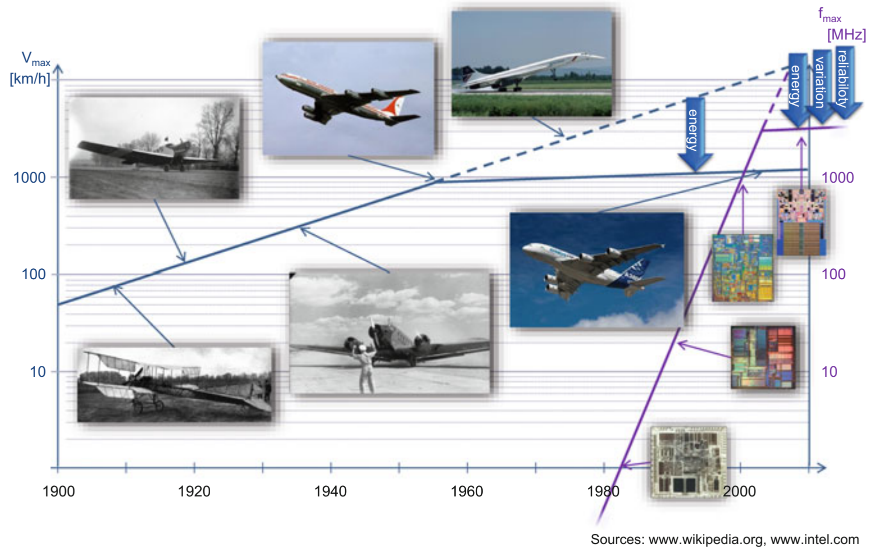


Fig. 3.1 Influence of extra-functional properties on the development of systems. Comparing trends in aircraft speed and clock frequency of general-purpose processors [21]

platforms at system level. In Sect. 3.4, the specification and implementation of a mixed-criticality multi-rotor use-case, integrating a safety-critical flight control algorithm with mission-critical payload processing on a single chip, is described. Section 3.5 describes the application of the CONTREX flow from Sect. 3.3 on the use-case presented in Sect. 3.4. This chapter closes with a conclusion and future work in Sect. 3.6.

3.2 The CONTREX Project

CONTREX [3] complements the above-mentioned existing European excellence through methodologies and tool support for the analysis and segregation along extra-functional properties, i.e., (real-) time, power, temperature, and reliability. These properties are the next roadblocks

1. to scale up the number of applications per platform and the number of cores per chip,
2. in battery-powered mobile platforms using wireless communication, and
3. for mission- and safety-critical systems when switching to technology nodes of 45 nm and below (see Fig. 3.2).

The CONTREX tools and design flow aim at cost-efficient and cost-sensitive design through analysis and optimization of power, temperature, and reliability with regard

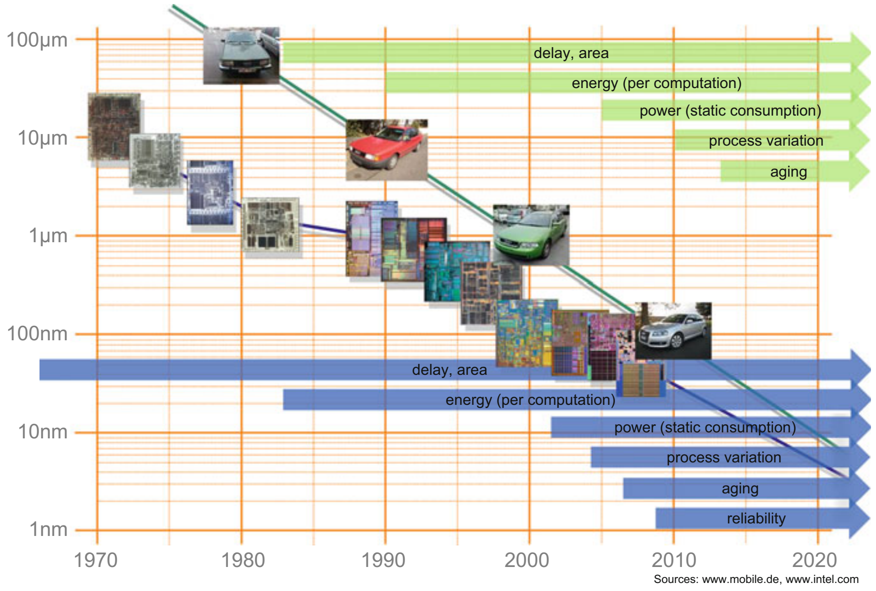


Fig. 3.2 Closing the technology gap between safety-critical and consumer computing platforms. Evolution and challenges in consumer electronics vs. electronic control units in cars [21]

to application demands at different criticality levels running on the same networked computing platform. The CONTREX approach will be integrated into an existing model-based design methodology [24, 25] and open source environments [16] that can be customized for different application domains and target platforms.

CONTREX puts its focus on the requirements taken from the automotive, avionics, and telecommunications domain and evaluates its effectiveness and integration into existing standards for the design and certification, based on three industrial demonstrators. An excerpt of the avionics demonstrator is show in Sect. 3.4.

To close the identified technology gap between custom designed mission- and safety-critical systems and cost-efficient platforms for consumer systems, the main goal of the project is to combine

– platform independent models for (control) applications with different criticalities, represented in domain specific modeling languages and formalisms,
– management and abstraction of multi-core hardware platforms' shared resources to guarantee temporal and spatial segregation for mixed-critical applications,
– management and abstraction of communication network resources to support temporal and spatial segregation to enable system-wide deployment and modularization in networked control applications, and
– cloud infrastructure abstraction and management techniques to support integration with data fusion/filtering for overall monitoring and online optimization of distributed large-scale control systems

with management and control of extra-functional properties, like power and temperature. These properties will limit the capabilities and realization of future ambient intelligent systems with regard to overall energy consumption, mobility (due to limited battery capacities), waste heat discharge, and finally reliability and availability. For this reason, the CONTREX project extends the industrial state of the art in mixed-criticality system design through a holistic design approach that considers extra-functional constraints as first-class citizens. It will represent and expose extra-functional properties under existing segregation and certification techniques (both in the design phase and during system operation), and finally include these properties into local (on the device/network node) and global (information exchange using cloud infrastructure) scheduling and control decisions.

The main goal of the project is to enable cost-efficient design, modeling, analysis, simulation, and exploration of complex networked control systems with mixed-criticality on different levels of abstraction. The project targets a meet-in-the-middle approach for the integration of existing design environments, models, and analysis and simulation tools. The project will extend the state of the art in domain specific control system modeling (top-down) through:

– Separation of design decisions for control application, deployment, and underlying hardware/software architecture at device level [24].
– Formalization, annotation, and refinement of constraints/contracts on extra-functional properties: time, power, and temperature [22, 23].

State-of-the-art segregation techniques for shared computing resources (i.e., multi-core systems) cover functional correctness and timing [2], but ignore possible influence and feedback paths originating from parasitic extra-functional effects [32]. Sharing the same computing platform (as shown in Fig. 3.3), multiple applications can interfere indirectly through power/energy and temperature properties. Running a hard real-time application and non-timing critical application (best effort) on the same execution platform (e.g., using a static Time Division Multiple Access (TDMA) scheduling), the non-timing-critical application can have an extra-proportional contribution to the overall power consumption. The increased power consumption heats up the whole chip and requires to slow-down (e.g., dynamic voltage and frequency scaling) dedicated cores and the memory subsystem to keep the chip temperature within a range allowing reliable operation. It has been predicted that at 22 nm, 21 % of a fixed-size chip must be powered off, and at 8 nm, even more than 50 % [6]. This dynamic reaction to control waste heat and reliability of the chip might have an influence on the core running the hard real-time task. This can be either directly through reducing the clock frequency of the core running the real-time application or indirectly through the effects in the memory subsystem. When designing such systems, all critical applications of the system need to designed with the explicit awareness of possible mode switches due to control of extra-functional properties.

Another example regarding the influence between mixed-critical applications and extra-functional properties can be found in mobile battery-powered systems. These systems suffer from limited battery capacities that keep running the system for a

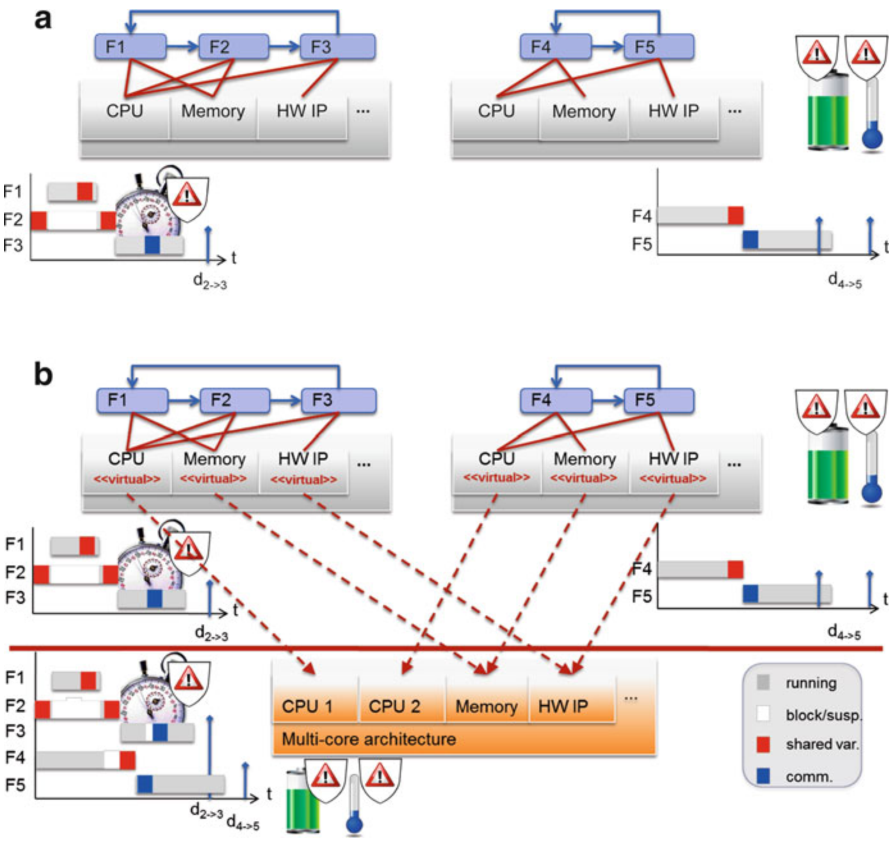


Fig. 3.3 Mixed-criticality systems now and then. (**a**) State of the art: Two control applications with different criticalities (*left*: hard real-time system with strict timing deadline, no power, or temperature constraints; *right*: soft real-time system with no strict deadline, hard power, and temperature constraints that might also originate from the systems environment/harsh operating conditions) implemented on two physical separated and/or distributed hardware/software platforms. (**b**) Future mixed-criticality systems: Two independent applications with different criticalities (from Fig. 3.3a) implemented on a multi-core hardware/software platform that enables temporal and spatial segregation (e.g., through static virtualization using TDMA). CONTREX enables analysis of real-time, power, and temperature properties and to implement segregation techniques regarding power consumption and heat dissipation

determinate amount of time. When running applications with mixed criticalities on mobile platforms the available battery capacity needs to be partitioned among applications and managed at system run-time to keep mission- and safety-critical services running for a determinate time (e.g., to reach the next re-charge cycle).

As pointed out, integrating mixed-criticality applications on multi-core and mobile battery-power computing platforms requires additional segregation along extra-functional dimensions, while keeping up classic temporal and special segregation properties. For this purpose, CONTREX extends state of the art in execution platform modeling and segregation for functional and extra-functional properties (bottom-up) through:

– Separation of physical hardware resources like processors, memories, and communication channels (on- and off-chip) from services that enable a transparent virtualization of different underlying hardware/software platforms,
– Characterization, abstraction, and explicit representation of timing, power, and temperature properties for specific hardware/software platforms, and
– Segregation along extra-functional dimensions.

The project combines top-down (control system modeling) and bottom-up (execution platform modeling) approaches in an integrated design environment establishing a missing link through:

– Deployment and mapping of control applications to a network of virtualized hardware/software platforms and network infrastructure abiding extra-functional properties.
– Simulation infrastructure that scales from detailed subsystem to overall networked control systems, including dynamicity of extra-functional properties.
– Support for the exploration of different deployment and mapping alternatives to obtain the most cost-efficient solution under the given extra-functional constraints.
– Cloud services for data acquisition and monitoring of extra-functional properties to obtain an overall health-state of the controlling system and the system under control including the coordination of global compensation actions at run-time.

In this chapter, the focus is on the extension of a virtual platform by extra-functional models for power and temperature. In this context a virtual platform is an executable model[1] of a real hardware platform that is capable to run the original software stack (full binary compatible) and supports tracing of functional and timing properties when the system is being executed.

Figure 3.4 depicts the main elements of the extended virtual platform. The *Virtual Platform* represents the functionality and the timing of the software stack (consisting of domain specific application, Operating System Services, Hardware Abstraction Layer services, and the processing elements). The *Extra-Functional Model* adds a power model for each processing element of the virtual platform. This power model is driven by the timing and activity of each processing element. If the operating frequency and voltage of the processing elements can be changed during run-time (e.g., Dynamic Voltage and Frequency Scaling (DVFS)), the power model receives this information from the HAL. The power model is tightly coupled with a temperature model for the thermal response of the package. Information about the local temperature can also be sent to a temperature sensor model that can be accessed by the software via HAL services to implement a dynamic power management. *(Extra-)functional Monitors* support the tracing and property checking

[1]Running on the host computer.

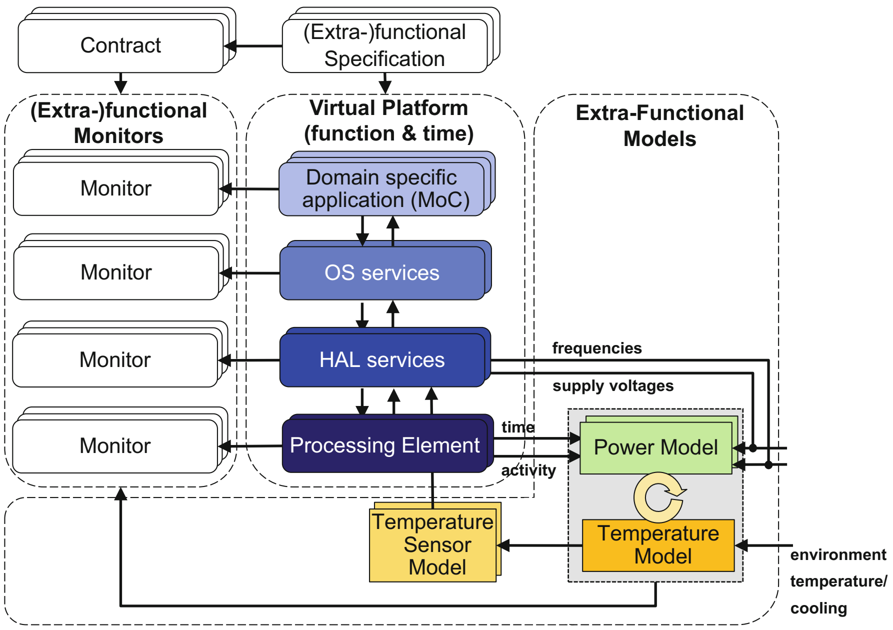


Fig. 3.4 Extended virtual platform

of function, power, and temperature over time intervals. These monitors are supported on the hardware platform level and all abstraction layers of the software stack. The following section provides more information about the combination of the virtual platform with the extra-functional models using timed value traces as uniform exchange format.

3.3 Considering Power and Temperature of a Full Chip at System Level—The CONTREX Flow

The design of embedded systems is typically constrained by extra-functional properties such as power, temperature, or degradation. Mobile embedded systems, for example, have in general a limited energy budget while avionic and aeronautic systems require a reliable operation with a long mean time to failure. In most cases, even multiple of the extra-functional properties are constrained and the design space needs to be explored during the design to fulfil all requirements.

Mixed-critical systems comprise tasks of different criticalities. If these are executed on a single platform, these tasks interfere with each other directly due to shared resources (e.g., shared memories) as well as through extra-functional properties. For the latter relation, mixed-critical systems introduce further degrees of freedom to the design space exploration that need to be explored. While safety-critical services need to operate at all time, mission- or non-critical services can be suspended during run-time (e.g., to reduce the dissipated power and temperature) in order to meet all requirements.

This work addresses the giant temporal and spatial granularity discrepancy in the analysis of the extra-functional properties power, temperature, and degradation for mixed-critical systems that highly interdependent and that cannot be regarded separately. From a temporal perspective, a power and temperature estimation can focus on seconds or minutes of operation to cover average and peak temperatures and power values. From a spatial granularity viewpoint, the analysis of an overall system at a coarse-grained level is sufficient for power and temperature to reflect electro-thermal coupling, boundary conditions, cooling capabilities, and energy budgets. In order to explore the design space and to meet all top-level requirements, every analysis of these extra-functional properties should provide estimation results within minutes of computation time. Thus, a scalable level of granularity is crucial and inevitable.

In the following, an extra-functional property estimation flow is described. The flow uses several existing point tools and addresses the above-mentioned problems. It uses a hierarchical composition of traces as the fundamental data structure and exchange format. These traces can be tailored to the needs of the different analysis steps via stream processing. The following paragraphs give an overview on the proposed flow describing its main elements, tools, and interfaces. The individual point tools and transformations used in the implementation of the flow are either commercially available, public domain, or result out of former research projects. Detailed references to the used tools are given in Sect. 3.5.

The overall flow is shown in Fig. 3.5. Green and red boxes indicate inputs and outputs to separate point tools and transformations (blue boxes). Red boxes thereby represent transient traces of the extra-functional properties. These traces are the fundamental data structure, as it will be described in the following.

The overall flow can be separated into three phases. First, the application is simulated on a virtual platform to gather power and activity data. Second, a temperature estimation is performed. An extension of this flow towards variation and aging is available at [10]. In parallel to these three subsequent parts, a contract satisfaction monitoring checks the validity of formally defined contracts on the extra-functional properties [22, 23].

3.3.1 Extended Virtual Platform Simulation

The basis for gathering the application dependent evolution of extra-functional properties over time is the execution of the application in a virtual platform that is extended by three means. First, power models are attached to the simulator defining possible power modes for each component that are characterized by a supply voltage, a clock frequency, an average switched capacitance per cycle, and average

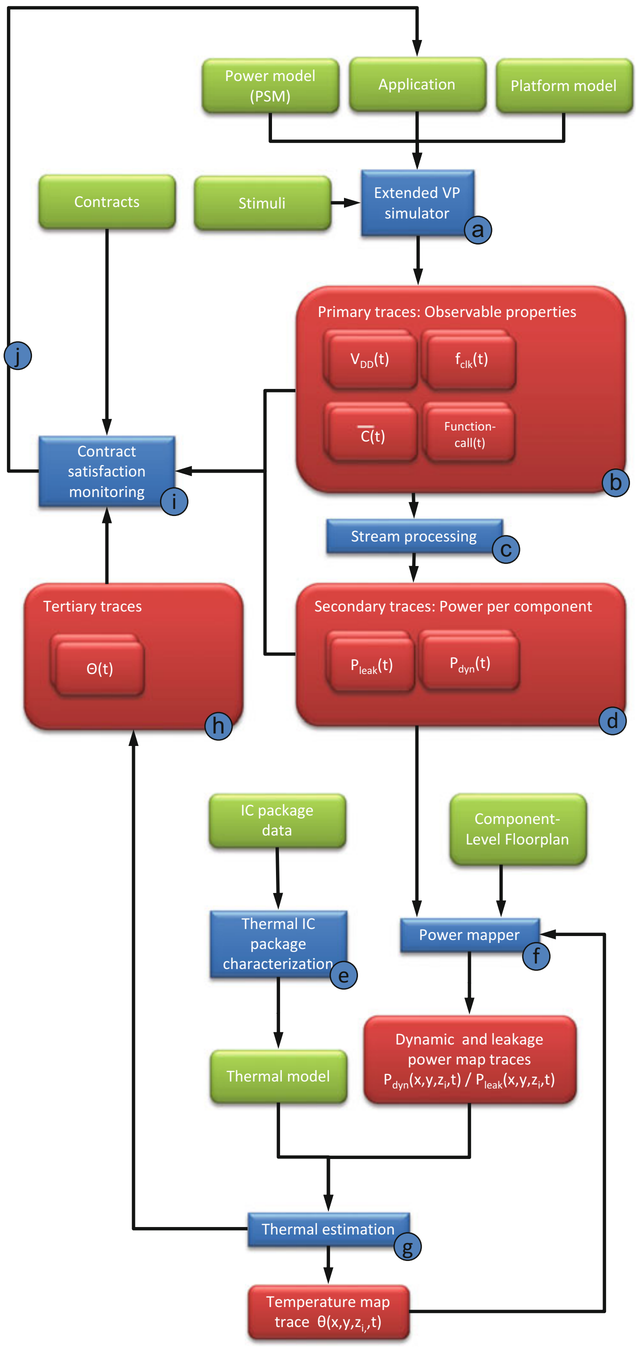

Power model (PSM)
Application
Platform model
Contracts
Stimuli
Extended VP simulator
a
Primary traces: Observable properties
$V_{DD}(t)$
$f_{clk}(t)$
$\overline{C}(t)$
Function-call(t)
b
j
Contract satisfaction monitoring
i
Stream processing
c
Tertiary traces
Θ(t)
h
Secondary traces: Power per component
$P_{leak}(t)$
$P_{dyn}(t)$
d
IC package data
Component-Level Floorplan
Thermal IC package characterization
e
Power mapper
f
Thermal model
Dynamic and leakage power map traces $P_{dyn}(x,y,z_i,t)$ / $P_{leak}(x,y,z_i,t)$
Thermal estimation
g
Temperature map trace θ(x,y,z_i,t)

Fig. 3.5 Overall extra-functional properties estimation flow

leakage currents [8]. Second, the simulator itself needs to have an introspection mechanism in order to be aware of power mode changes that may be taken either automatically due to inherently implemented power management policies or due to sleep state transitions that are triggered by the application. Third, the simulation is extended by a tracing engine that protocols the extra-functional properties on appearance as described in the following.

3.3.2 Primary Traces: Observable Properties

Primary traces are transient traces of observable properties that can be directly observed during the execution of an application on the extended virtual platform. The purposes of theses traces are to derive the dissipated power over time at a component-wise level in order to feed a subsequent transient temperature estimation and to establish a link between the application execution and its power consumption to validate if defined constraints on the power budget are met. Thus, the primary traces need to cover every parameter that is necessary to compute the power consumption trace per component and to link it with application execution:

- **Transient supply voltage traces per component:** Multiple voltage domains in a heterogeneous SoC (e.g., different cores, DSPs, or IO blocks) can operate at different supply voltages that may even change over time due to power management techniques. A single voltage domain can cover multiple components.
- **Transient clock frequency traces per component:** Power management techniques can be applied at component level and often rely on a reduction of the clock frequency. Thus, the clock frequency needs to be traced in order to derive the power consumption.
- **Transient switched capacitance per component:** The switched capacitance reflects the activity of the application and is the fundamental primary parameter for dynamic power.
- **Transient leakage currents per component:** Leakage currents also depend on power management techniques and are the fundamental parameter for the static power consumption of a component.
- **Transient function call traces per component:** To establish a link between the power consumption and the application a tracing of function calls is an adequate way. Further, it enables the use of design space exploration methods.

3.3.3 Stream Processing

The primary traces of observable properties are transformed to power consumption traces by a stream processing. This stream processing is defined on the traces and its implementation is type safe [9]. Figure 3.6 shows exemplary traces of the

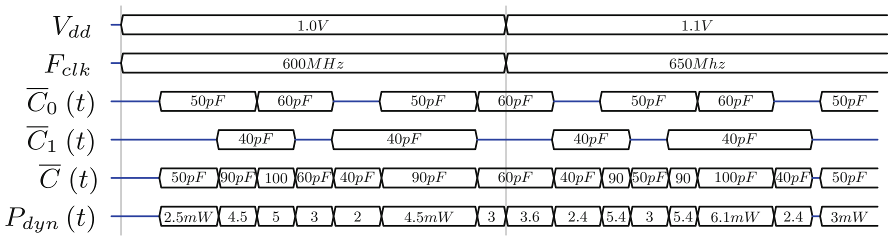


Fig. 3.6 Traces of supply voltage, clock frequency, cycle-averaged switched capacitances, and the dynamic power trace as it can be a target of a stream processor [9]

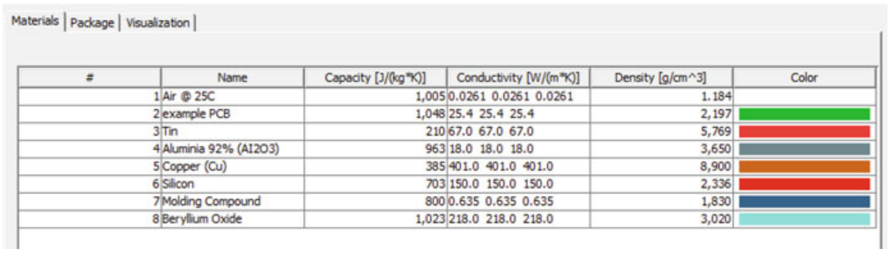
supply voltage of a single voltage island, a clock frequency trace, the average switched capacitances per cycle $\overline{C_0}$ and $\overline{C_1}$ for two independent components, and the aggregated averaged switched capacitance per cycle trace as well as the overall dynamic power trace. In the middle of the trace time, the voltage and frequency are scaled as it is commonly done by DVFS techniques. The latter two traces can be derived by simple stream processors. While the aggregated switched capacitance per cycle is derived by summing up the capacitances of the two components, the stream processor to compute the dynamic power needs to know the fundamental physical relation $P_{dyn}(t) = C(t)V(t)^2F_{clk}(t)$ in order to aggregate the streams.

3.3.4 Secondary Traces: Power per Component

Secondary traces describe the power consumption of each hardware component. The stream processing distinguishes between dynamic and static power to be able to consider the temperature dependency in the leakage power traces.

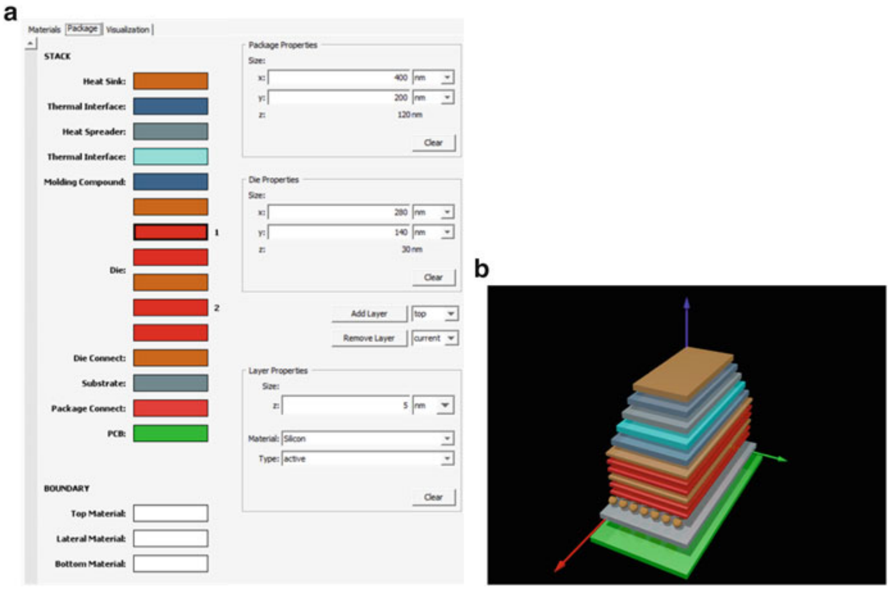
3.3.5 Power Mapping

In order to feed the subsequent temperature simulation a two-dimensional power map trace needs to be generated. Therefore, the component-level traces are mapped to a component-level floorplan. In this step, the power is averaged to the total component area because neither the power consumption nor the place and route data is available for commercially available SoCs at a more fine-grained level. The power mapping results in two map traces: one for the dynamic power and one for the leakage power distribution.


Materials | Package | Visualization

#	Name	Capacity [J/(kg*K)]	Conductivity [W/(m*K)]	Density [g/cm^3]	Color
1	Air @ 25C	1,005	0.0261 0.0261 0.0261	1.184	
2	example PCB	1,048	25.4 25.4 25.4	2,197	🟩
3	Tin	210	67.0 67.0 67.0	5,769	🟥
4	Aluminia 92% (Al2O3)	963	18.0 18.0 18.0	3,650	⬜
5	Copper (Cu)	385	401.0 401.0 401.0	8,900	🟧
6	Silicon	703	150.0 150.0 150.0	2,336	🟥
7	Molding Compound	800	0.635 0.635 0.635	1,830	🟦
8	Beryllium Oxide	1,023	218.0 218.0 218.0	3,020	🟦

Fig. 3.7 Material properties



Package Properties
Size:
x: 400 nm
y: 200 nm
z: 120 nm

Die Properties
Size:
x: 280 nm
y: 140 nm
z: 30 nm

Add Layer: top
Remove Layer: current

Layer Properties
Size:
z: 5 nm
Material: Silicon
Type: active

Fig. 3.8 IC package model. (**a**) IC properties and material stack. (**b**) Intersection through IC package

3.3.6 Thermal Model Generation of IC Package

Beside the power map traces, a thermal estimation requires a compact thermal model of the targeted IC package. This model reflects the geometry, used materials and their thermal properties, and the environment such as applied active or passive cooling measures [28].

Figure 3.7 shows an excerpt from a material database defining all physical material properties that are relevant for a thermal estimation: thermal capacity, conductivity in different directions, and density. Based on this database, an IC package is defined as shown in Fig. 3.8a. It covers the die stack that is embedded in the surrounding package materials as well as the connection to the PCB, the PCB itself, and the boundary conditions.

Figure 3.8b shows an intersection through a sample IC package showing the stack of different materials as it is defined in Fig. 3.8a.

Based on this IC package description the compact thermal model is created that is used in the following thermal estimation. The characterization step needs to be done only once for each targeted IC package.

3.3.7 Thermal Estimation

A transient thermal estimation reads in the thermal model of the IC package and the power map traces to stimulate the simulation. It computes a temperature distribution map that evolves over time. Further, it also outputs aggregated temperature traces at the component-level granularity that only contain the minimum, maximum, and average temperature for each component since constraints can be made on these properties. During the thermal estimation, electro-thermal coupling due to temperature-dependent leakage currents is reflected.

3.3.8 Tertiary Traces

While primary traces can directly be observed during the virtual platform execution and are transformed to secondary power traces via stream processing, component-level temperature traces result out of downstream analysis steps and are denoted as tertiary traces. Each temperature trace describes the minimum, maximum, and average temperature per component that can be observed in the occupied area of the component within the floorplan. From a timing perspective, the temperature traces are directly correlated to the simulated execution time.

3.3.9 Contract Satisfaction Monitoring

After or even during the overall estimation flow, a monitor checks if constraints on the traced extra-functional properties are hold. These constraints are defined in a formal contract specification with an assume/guarantee semantics [22, 23]. For each specified contract, a monitor observes the affected primary, secondary, or tertiary trace and raises an exception if the assumption (precondition) is violated. This contract satisfaction monitoring checks the execution trace of the application dependent on its stimuli against the formulated contracts in a simulative manner and does not perform a formal verification. Examples of such checks are threshold temperature values, overall power budgets, or peak power consumptions that should not be exceeded.

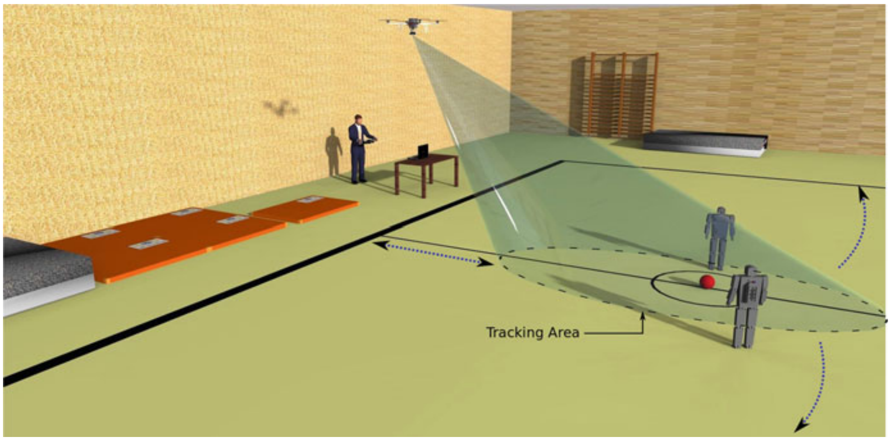


Fig. 3.9 Remotely piloted multi-rotor scenario: Ball detection and tracking during a match [1]

3.4 A Mixed-Criticality Use-Case

This section briefly presents the specification and implementation of a mixed-criticality multi-rotor system: integrating a safety-critical flight control algorithm with mission-critical payload processing on a single chip [1, 26, 30]. The use-cases are based on an existing multi-rotor platform, which consists of a chassis, four rotors, the motor driver hardware, and a remote control [11].

3.4.1 Selected Scenario

The aim of the presented use-case scenario is the definition of a mixed-criticality system with a reasonable compute intensive mission-critical application. In the selected scenario, this mission-critical application is an on-board video processing-based detection and tracking of a ball in a robot soccer match, as shown in Fig. 3.9.

While a pilot controls the position of the multi-rotor system, an object-tracking algorithm scans the video images for the ball. The camera is mounted on a gimbal to be able to adjust its view angle autonomously by an on-board control algorithm to center up the ball. The algorithm to detect the ball determines objects with a pre-configured color of every shape. Since the multi-rotor system is used in a gym, the ball needs to have a distinct color for the image filter algorithm. Therefore, a magenta colored ball was chosen. The video images of the camera are streamed to a ground control station for display.

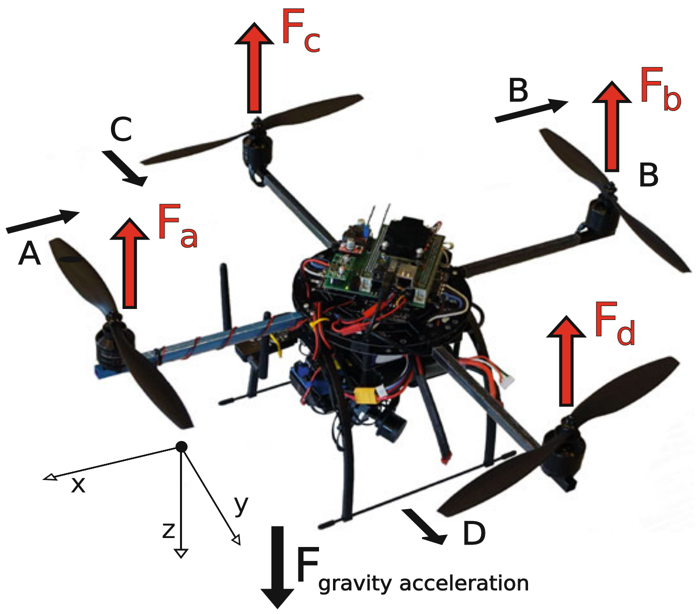


Fig. 3.10 Used multi-rotor system (Quadcopter) [29]

3.4.2 Fundamentals

Multi-rotor systems are helicopter-like flying platforms with usually four or more rotors. The rotors are fixedly mounted on the rigid frame and the airflow is unidirectional in the direction of the ground. The used multi-rotor system is shown in Fig. 3.10.

Since the only movable parts of the system are the rotors, it can solely be controlled via the forces generated by different engine speeds. A higher/lower engine speed will generate a bigger/smaller force at the particular frame arm. Figure 3.10 gives in addition an overview of the forces, which are generated at the mounting points and a naming of the rotors for the following explanations. To move the multi-rotor system up or down along the z-axis, the engine speeds of all rotors have to be lowered or raised equably. To rotate the system around the z-axis, the torque between the rotors has to be unbalanced. Normally, the torque would be balanced while two engines are turning clockwise and the other two counter-clockwise. However, if the engine speed of rotor *D*, *C* is raised/lowered and the engine speed of rotor *A*, *B* is equably lowered/raised, the system will turn clockwise/counter-clockwise around the z-axis. To move the multi-rotor system along the horizontal x- and y-axes, it needs to be pitched and rolled. By raising the engine speed of rotor *B*/*A* and lowering the speed of rotor *A*/*B*, the system will pitch and move along the x-axis in direction of rotor *A*/*B*. The same applies for the y-axis.

Since multi-rotor systems are not automatically stabilizing, they need to be controlled at a high frequency. To handle this, sensors for calculating the attitude and altitude of the system are attaches to the avionics and control algorithms are stabilizing it.

3.4.3 Payload Setup

To realize an object-tracking application, which is able to track a ball, a motor-driven camera gimbal and a small high definition camera mounted on the bottom of the multi-rotor system as payload are used. The gimbal has the ability to turn around all three axes. In that way, it is possible to settle all rotational movements of the system to get a stable and horizontally aligned video image. The axes are adjusted by three brushless motors, which are driven by an extra controlling unit. The controlling unit gets the set points for the attitude and movements of the camera to manage the field of vision. A USB Wi-Fi stick is connected to the payload setup to communicate with the ground control station.

3.4.4 Hardware

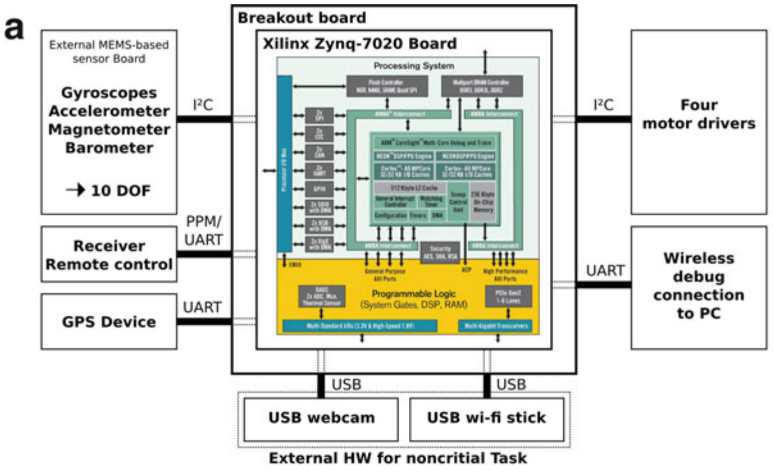
The hardware, which is mounted on the top of the flying platform, is the centerpiece of the multi-rotor system. It consists of PCBs, the processing system, sensors, interfaces, and an own power supply.

The heart of the hardware is a multiprocessor system-on-chip (MPSoC) represented by the Zynq 7020. The Xilinx Zynq 7000 MPSoC family [35] combines a dual core processor and programmable logic. The Zynq 7020 consists of a dual ARM Cortex-A9 MPCore at 866 MHz and an Artix-7 FPGA with 85k logic cells.

An AMBA Interconnect connects the ARM dual core to the peripherals. There are plenty of available interfaces, which can be connected to the pinout of the MPSoC by the Processor I/O MUX. The AMBA Interconnect represents also the interface to multi-port DRAM Controller and the Flash Controller as well as the connection to the Programmable Logic (FPGA) part of the MPSoC. With the Artix-7 FPGA it is possible to define and build further interfaces, processing elements (e.g., MicroBlaze softcores) or also specialized hardware for the payload processing tasks.

The hardware structure consists of three PCBs that are stacked together. The PCB stack is shown in Fig. 3.11 and is composed of the mainboard, the carrier board, and the Zynq board (see Fig. 3.11c).

The sensors are mounted at the mentioned mainboard and are used to determine the current attitude and the current altitude of the multi-rotor system. We use a 9 Degree-of-Freedom (DoF) MPU9150 from InvenSense Inc. and a 1 DoF BMP085 from Bosch. The MPU9150 sensor package contains four different types of sensors:



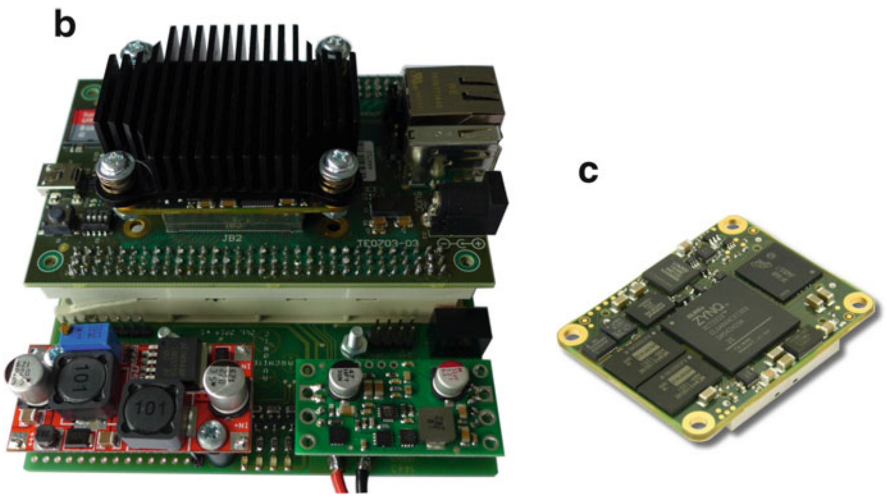


Fig. 3.11 PCB stack and components of the multi-rotor system. (**a**) Overall hardware architecture. (**b**) PCB stack [29]. (**c**) TE0720-02-2IF Zynq board (hidden by heat sink) [31]

- 3-axes gyroscope for measuring the angular velocities around the x-, y-, and z-axis;
- 3-axes accelerometer for measuring the velocities along the x-, y-, and z-axis;
- 3-axes magnetometer for measuring the earth magnetic field along the x-, y-, and z-axis;
- and a temperature sensor.

The MPU9150 sensor is responsible for a stable flight behavior of the overall system. The BMP085 sensor contains a barometric sensor and a temperature sensor.

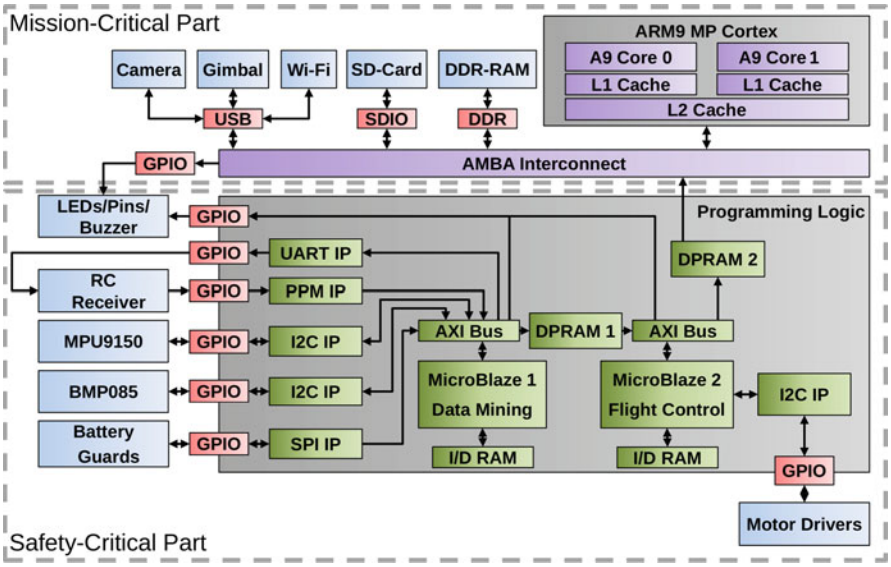


Fig. 3.12 Developed architecture on Zynq MPSoC [29]

Both sensors have an I^{2}C interface to communicate with the processing system. To determine the attitude the gyroscopes, accelerometers, and magnetometers are used. To determine the altitude the values of the accelerometer and the barometer are fused. The barometer can be used to calculate the height, because the pressure is approximately proportional to the height above ground level, under the assumption that the weather is not changing while flying.

The Mixed-Critical Architecture consists of a processing architecture on the ZNYQ that uses the ARM cores and the programmable logic to implement additional processing elements, interfaces, memories, and other required components. This architecture is able to serve as execution platform for mixed-critical tasks, in the sense that the safety-critical tasks must not be influenced by any other task. In the presented multi-rotor system, two main tasks are executed: The safety-critical flight algorithms and the mission-critical video processing. To realize this, the Zynq is divided into two separate parts. The ARM dual core is used to process the mission-critical tasks using a Linux operating system. The safety-critical flight algorithms are executed bare metal, without an operating system, on two dedicated MicroBlaze processor instances, each with local dedicated RAM, in the FPGA part of the MPSoC. The resulting block-level implementation of the architecture with interfaces and connected external systems is shown in Fig. 3.12.

The figure is divided into two parts. On the top all used components of the mission-critical part are located, at the bottom all components of the safety-critical part are located. The mission-critical part uses the ARM dual core as processing elements. The communication with the external components, like the

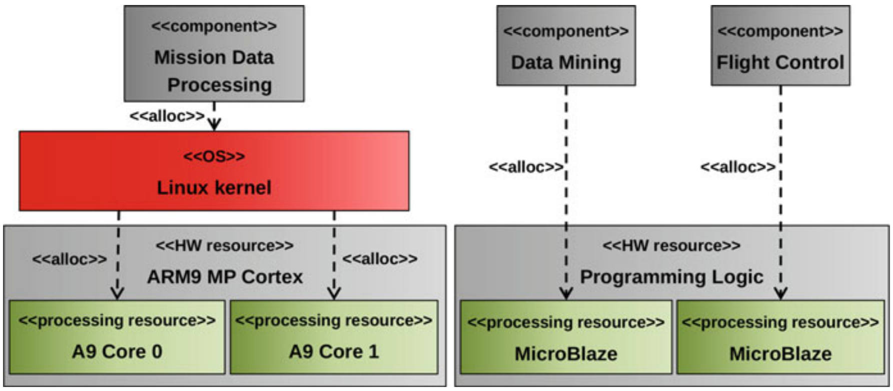


Fig. 3.13 Mapping of the software [29]

camera, the gimbal, the Wi-Fi stick, the SD-Card for loading the software image and logging some telemetry information, and the DDR-RAM, is realized via the AMBA interconnect and built-in interfaces. At the bottom, all needed parts for the safety-critical flight algorithms can be seen. For the processing, two MicroBlazes are used. *MicroBlaze 1* integrates all sensor inputs, which are located on the left side. *MicroBlaze 2* uses its resulting data to calculate the final control values of the motor drivers. Communication between the MicroBlazes is realized by the dual-ported RAM *DPRAM 1* where *MicroBlaze 2* has read-only access, to setup a unidirectional communication (FIFO mode). The same for *DPRAM 2*. The mission-critical part has solely read-only access to the memory, to get the telemetry data. This way, the mission-critical part is not able to influence the safety-critical flight algorithms. The *RC Receiver* transmits the control values of the remote control as Pulse-Position Modulation (PPM) encoded one wire signal, which is decoded by an IP core and received by *MicroBlaze 1*. This MicroBlaze transmits also telemetry data via an UART interface back to the remote control. Like mentioned before the sensors MPU9150 and BMP085 are connected to the FPGA by I^{2}C interfaces and also read out by *MicroBlaze 1*. The *Battery Guards* are also handled by this processing element via a SPI interface. All processing elements have connection to separate LEDs, Pins, or Buzzers for debugging purposes. In the diagram, they are summarized up as a single block.

3.4.5 Software

The software of the whole system consists of three main parts, two for the safety-critical task, which are executed on the two MicroBlazes and one for the mission-critical application. The mapping of these tasks are shown in Fig. 3.13. The Zynq MPSoC processes both the safety-critical and the mission-critical part of the software, which will be described in the following.

3.4.5.1 The Safety-Critical Part

of the software is completely mapped to the MicroBlazes in the programmable logic part of the MPSoC. The data miner is responsible for the whole sensor processing. This means, it is getting all sensor data as inputs, filters these data, and makes all needed calculations to get the final attitude, altitude, and control values for the second component as a result. In addition, the data miner processes the telemetry data for the remote control and some debug information, which is output via the GPIOs. The flight control system consists of an attitude and altitude Proportional-Integral-Derivative (PID) controller. Based on the internal state (e.g. attitude and altitude) and the command values (e.g. navigation set point from the remote control), the flight controller computes new torque values for each of the four motor drivers to stabilize and navigate the system. In addition, this component processes some debug data, like reading the status of the motor drivers (including temperature and overload information) and provides the data, which is needed by the mission-critical application.

3.4.5.2 The Mission-Critical Part

is executed on the ARM processing system of the MPSoC. A Linux kernel is used as operating system for handling the different processes and needed interfaces of the tasks. Object detection is activated by the remote control. Whenever active, the current video frame of the camera is continuously grabbed. After grabbing a frame, it will be provided for the ground control station by the video streaming server. Next to this service, the video frame is downsized for the object detection to reduce the workload and to increase the possible frame rate for detecting the object. The object detection searches for the occurrence of a pre-configured color. A color filter is used, which creates a binary image. In this image, the position of the object with the pre-defined color is identified. The coordinates of the object in the image are used afterwards to calculate the new set points for the gimbal controller to update the camera angle.

3.4.6 Ground Control Station

The ground control station (GCS) is represented by a simple graphical user interface (GUI) which connects to the multi-rotor system via the Wi-Fi connection and gets all needed data. Telemetry data, like attitude, altitude, battery voltages, temperatures of the system, and control values, are displayed also as the state of the mission-critical application. In addition, the video stream of the on-board camera is shown in the GUI. The ground station's GUI only displays data and has no control access to the multi-rotor system.

3.5 Application of the CONTREX Flow

This section exemplifies the application of the CONTREX flow (described in Sect. 3.3) on the mixed-criticality use-case of the previous section.

3.5.1 Virtual Platform

The basis for gathering the application dependent evolution of extra-functional properties power and temperature over time is the execution of the application in a virtual platform.

Open Virtual Platforms™(OVP™) [13] by Imperas™provides a complete simulation infrastructure for virtual platforms of embedded systems. OVP consists of three main components: OVP APIs to build own models and to create extensions for the simulator, a library that contains many open source processor and peripheral models that are freely available, and OVPsim™which is the simulation kernel to execute these models. Since OVP only performs functional simulations, it enables very fast simulation speeds. This is done by using code morphing from the virtualized processor's instruction set to the host processor's x86 instruction set. The virtual platforms simulated by OVPsim are observable via the APIs and debuggers that can be connected to the processor models. In that way, OVP can be used as a development environment for embedded software.

Figure 3.14 shows the Zynq™-7000 SoC Extensible Virtual Platform [33] used as core functional simulator for the Xilinx Zynq MPSoC. It consists of the ARM® Cortex™-A9 MP OVP processor model, functional models of most Zynq-7000 SoC peripheral devices, and Transaction Level Interfaces (TLM 2.0) for integration of custom devices within the programming fabric. The safety-critical part of the use-case hardware platform (see Fig. 3.12), consisting of two Xilinx MicroBlaze OVP processor models, has been integrated using this TLM interface. As shown in Fig. 3.13, the mission data processing (i.e., object tracking) is running on Linux (in SMP mode) on the ARM Cortex-A9 MP system. The data mining and flight control algorithms are running bare metal, each on one MicorBlaze processor.

After establishing a functional software binary compatible virtual platform of our MPSoC platform, the following extensions to the virtual platform are preformed:

Timing Models defining a cycle approximate[2] progress of the functional simulation.

Power Models defining possible power modes for each component that are characterized by a supply voltage, a clock frequency, and activity metrics.

Tracing Engine that protocols the extra-functional properties on appearance as described in the following.

[2]While the MicroBlaze timing model is close to clock cycle accuracy, the ARM timing model is working on a coarse-grained instructions per cycle level.

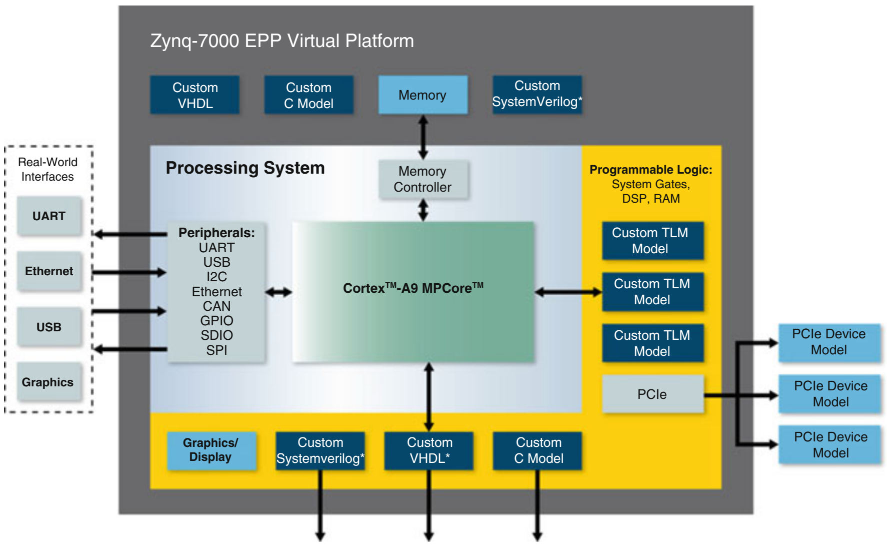


Fig. 3.14 Zynq™-7000 SoC Extensible Virtual Platform [33]

Within CONTREX, the OVP Simulator [13] will be used and adopted offering an extended API via the M*SIM extension [12] (Fig. 3.15).

3.5.2 *Timing Model*

OVPsim is a functional platform simulator, which uses binary translation to reach its high simulation performance. The simulated time of the platform is estimated by using an instructions per second metric of the processor model. This way the simulation kernel counts the executed instructions and divides them by the instructions per second metric to give an approximate timing of the simulation. This approach is not very accurate since it completely neglects pipeline and memory subsystem effects. For getting a more realistic estimation of the executed cycles and the timing of the virtual platform, a quasi-cycle accurate timing model based on [27] has been developed.

The timing model uses the OVPsim Innovative CpuManager Interface (ICM) API. This API offers functions to get the needed run-time information from the virtual platform. The core idea of the timing model is to observe all executed instructions of the MicroBlaze softcore and calculate the overall needed cycles of each one. For this purpose, the timing model is divided into three parts:

1. Instruction and Event Sniffer (IES)
2. Timing Estimator (TE)
3. Pipeline Model (PM)

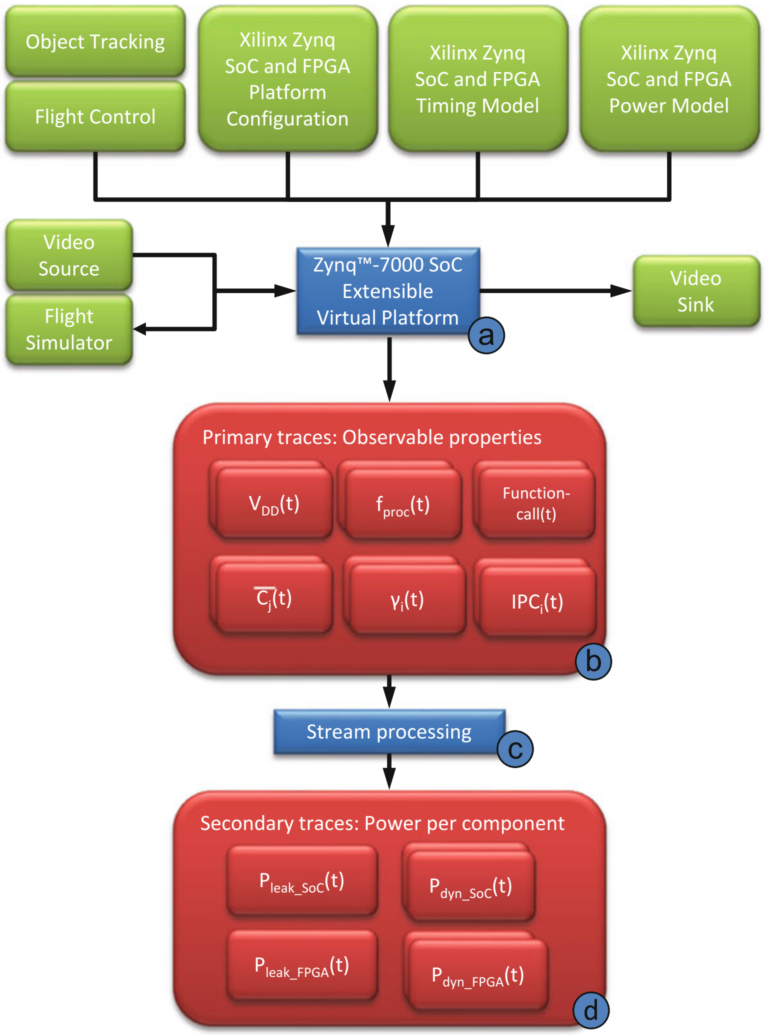


Fig. 3.15 Application of the virtual platform power model on the use-case

The IES uses the ICM API to register callbacks, like an instruction fetch callback, for tracing all executed instructions of the MicroBlaze model or a read callback of the timer counter register, to analyze the behavior of the virtual platform's timer model. The IES calls the TE for every fetched instruction.

The TE uses the operand code of the fetched instruction and uses a look-up table to get its number of processor cycles. As the MicroBlaze contains an instruction pipeline with five stages, a pipeline model was implemented to improve the accuracy of the timing model.

The PM also gets the operand code and looks up the forwarding stage of the instruction's result. This information is stored in a software pipeline. To compute the pipeline stall cycles, register dependencies are searched within the present and the four last fetched instructions. The pipeline stall cycles are returned by the PM to the TE that computes the total cycles of the present instruction, as well as the global cycles processed by the MicroBlaze. The TE returns all information to the IES that computes the system time and updates the simulation time of the virtual platform via the ICM API.

3.5.3 Power Model

Since the Xilinx Zynq is a very complex system that is composed of many single components in one package, also its power consumption depends on many different factors. To estimate the power consumption of the Zynq platform, the Xilinx Power Estimator (XPE) [34] can be used. It is an Excel spreadsheet that is able to compute the overall power consumption of the Zynq system based on a given static configuration. Since this power model cannot be used in a dynamic execution environment, i.e., virtual platform, a power model that is based on the XPE has been developed. The power model is divided into two parts:

Zynq SoC consisting of the ARM dual core, memory subsystem, and peripherals.
Zynq FPGA consisting of two MicroBlaze processors, Block-RAM and peripherals implemented in the FPGA.

The SoC dynamic power model input parameters are:

- clock frequencies of the ARM cores, the memory, the AXI bus, and the I/O ports (in MHz each)
- number of used cores (0, 1 or 2)
- bandwidth of the AXI bus (32 or 64 bit)
- load of the ARM cores (0-1)
- read and write rates of external DDR3 memory (0-1 each)
- load of the AXI bus (0-1)

Many of these information is available by the configuration of the Xilinx Vivado Project for the given platform, like the number of used ARM cores, their clock frequency, as well as the bandwidth and clock frequency of the AXI bus or the clock frequency of the I/O ports. The utilization parameters (which have to be given in percent) have to be estimated by the given application that is executed by the processing system. Further details of this power model are omitted. Instead, we refer to a more simple power model [15] that computes the ARM A9 dual core power model in mW:

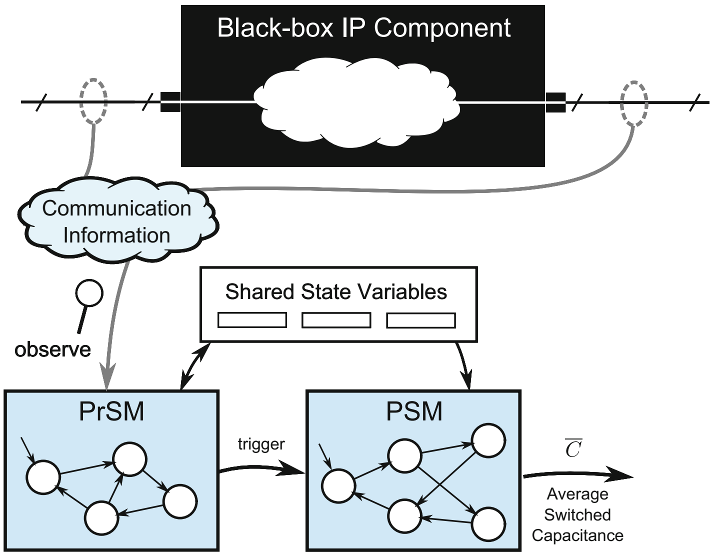


Fig. 3.16 PSM approach overview for non-invasive simulation of energy consumption

$$P_{\text{dyn\_SoC}}(t) = 0.63 \cdot f_{\text{proc}}(t) + 1.4 \cdot \gamma_1(t) + b \cdot \sum_{i=1}^{2} \gamma_{2_i}(t) + c \cdot \sum_{i=1}^{2} \text{IPC}_i(t) + 12.45 \qquad (3.1)$$

with:

- f_{proc} processor clock frequency in MHz
- γ_2 L2 cache miss rate ($0 \leq \gamma_2 \leq 100$)
- γ_1 L1 cache miss rate ($0 \leq \gamma_1 \leq 100$)
- IPC Instructions Per Cycle ($0 \leq \text{IPC} \leq 2$).

The FPGA dynamic power model of the safety-critical part is dominated by the switching activity of the two MicroBlaze processors and the involved Block-RAM resources (for local data and instruction memory and for the unidirectional communication between the MicroBlaze processors and the ARM SoC subsystem). For this reason, a Power State Machine characterization for each MicroBlaze processor and peripheral implemented inside the FPGA has been chosen.

Figure 3.16 gives an overview of our I/O observation-based approach for annotating state-based power information at hardware components. This approach is also well suited for black-box IP models, such as the MicroBlaze processor.

Selected I/O ports of an IP component are observed over time to approximate the internal functionality. Based on these observations, a *Protocol State Machine* (PrSM) is controlled. The main task of the PrSM is to trigger state transitions in the *Power State Machine* (PSM) based on the observation and interpretation of the interaction between component and environment. The state of the PSM represents

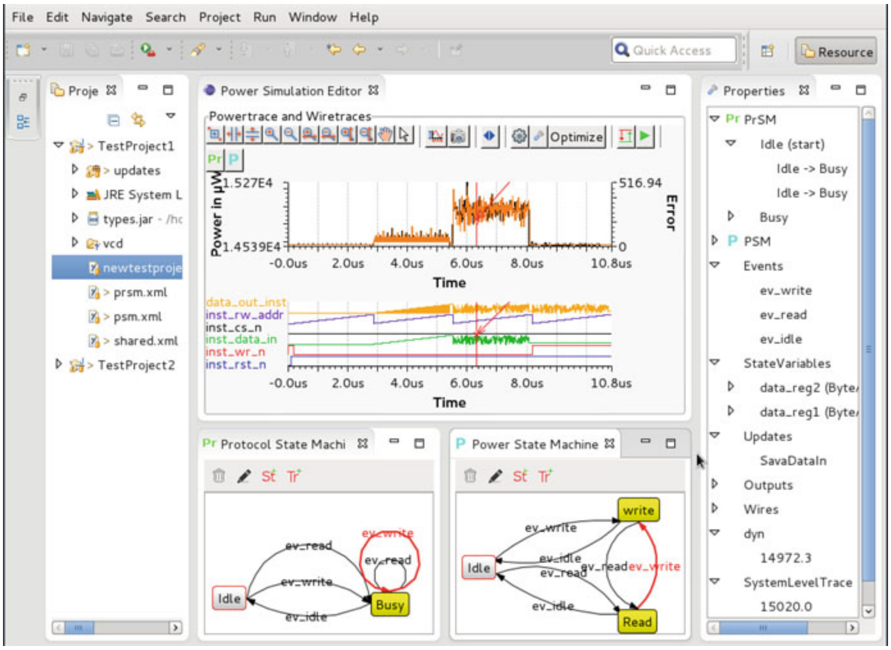


Fig. 3.17 PSM Eclipse plug-in GUI [19]

the average switched capacitance $\bar{C}$ of an IP component. By using the capacitance instead of power, the output of the PSM is independent of Dynamic Voltage and Frequency Scaling (DVFS) parameters such as supply voltage and clock frequency.

By modeling the interdependencies between I/O and internal states, the PrSM extracts the energetic relevant events to orthogonalize the communication and functional artifacts of the non-functional PSM model. This may lead to a reduction of complexity in the PSM because it only describes the different internal operation modes, whereas the PrSM covers the access protocol of the component. Furthermore, the separation of PrSM and PSM has the advantage that components with the same access protocol and different internal implementations could use the same PrSM, only the PSM has to be changed.

PSM and PrSM are each modeled as an Extended Finite State Machine (EFSM), which allows the extension of a simple FSM with (shared) state variables to reduce the complexity. More details about this state-based power modeling approach can be found in [17, 18, 20].

Figure 3.17 shows our PSM Eclipse plug-in GUI with the PrSM and PSM of a simple memory. It has been created based on a power measurement trace. From this plug-in, an executable SystemC model of the PrSM and PSM can be generated and used in a SystemC TLM 2.0 simulation environment of the virtual platform.

The static power consumption (leakage) of the Zynq is only supply voltage dependent and currently represented by a technology dependent leakage current I_{leak}.

$$P_{\text{leak\_SoC}} = V_{\text{DD\_SoC}} \cdot I_{\text{leak\_SoC}} \quad P_{\text{leak\_FPGA}} = V_{\text{DD\_FPGA}} \cdot I_{\text{leak\_FPGA}} \tag{3.2}$$

In Eq. (3.2) $V_{\text{DD\_SoC}}$ is the supply voltage of the Xilinx Zynq processor system and $V_{\text{DD\_FPGA}}$ is the supply voltage for the Xilinx Zynq programmable logic (FPAG) part.

Consideration of temperature-dependent static power consumption [14] is subject of future work. The source of static power consumption is a combination of sub-threshold I_{sub} and gate-oxide leakage I_{ox}.

$$I_{\text{leak}} = I_{\text{sub}} + I_{\text{ox}} \tag{3.3}$$

$$I_{\text{sub}} = K_1 \cdot W \cdot e^{-V_{\text{th}}/n \cdot V_{\Theta}} \left(1 - e^{-V/V_{\Theta}}\right) \tag{3.4}$$

K_1 and n are experimentally derived, W is the gate width, and V_{Θ} in the exponents is the thermal voltage. At room temperature, V_{Θ} is about 25 mV; it increases linearly as temperature increases.

$$I_{\text{ox}} = K_2 \cdot W \cdot \left(\frac{V}{T_{\text{ox}}}\right)^2 \cdot e^{-\alpha \cdot T_{\text{ox}}/V} \tag{3.5}$$

K_2 and α are experimentally derived. T_{ox} is the oxide thickness.

3.5.4 Temperature Model

Figure 3.18 shows the application of the virtual platform temperature model on the use-case.

In order to feed a subsequent temperature simulation a two-dimensional power map trace needs to be generated. Therefore, the component-level traces are mapped to a component-level floorplan as it is visualized in Fig. 3.19a. In this step, the power is averaged to the total component area because neither the power consumption nor the place-and-route data is available for commercially available SoCs at a more fine-grained level. The power mapping results in two map traces: one for the dynamic power and one for the leakage power distribution. Figure 3.19b shows the described mapping of power traces to components as it is done in Docea Power's Aceplorer tool [4]. It supports multiple formats of traces such as VCD and CSV. Furthermore, it supports equations that can be entered as power model to cover the electro-thermal coupling.

Similar to the floorplan, a package description for the ZYNQ SoC has been realized in Docea Power's Thermal Profiler [5].

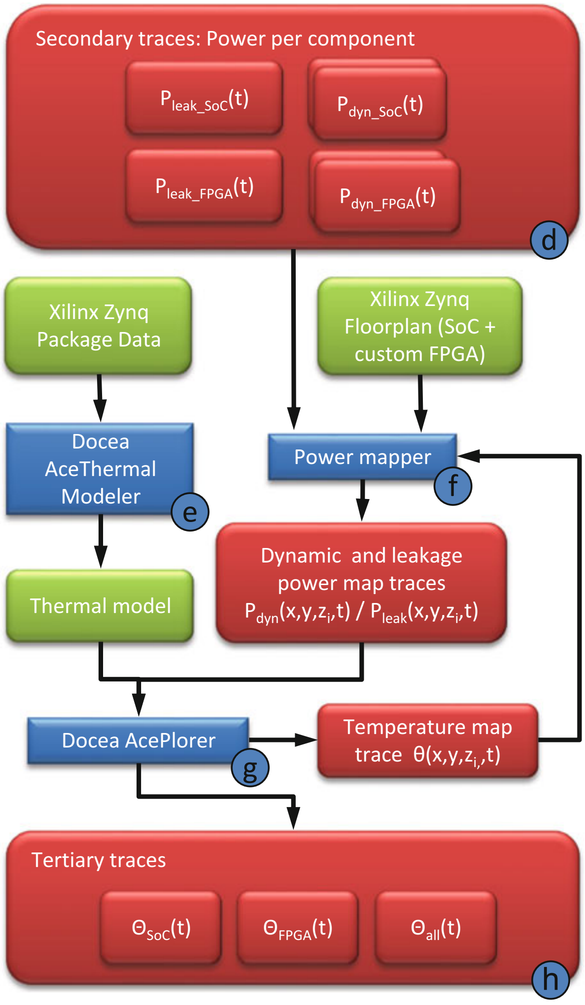
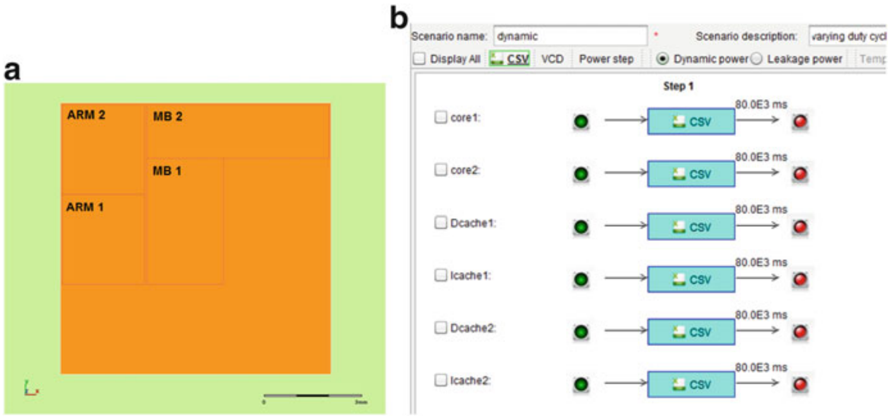


Fig. 3.18 Application of the virtual platform temperature model on the use-case

Figure 3.20 shows the example of a transient power and temperature trace of a single component over time. It visualizes how the temperature follows the power consumption. In Fig. 3.21 a power and temperature over time trace of the multi-rotor use-case is shown. It depicts the power consumption for each ARM core and the combined power consumption of the MicroBlaze cores. The temperature over time trace is also shown for each component. In the depicted scenario, the set threshold temperature of 65°C for the MicroBlazes in the FPAG has been exceeded. A possible solution to keep the MicorBlaze processor temperature under the given threshold is a reduction of the workload on the ARM processors that generate the main heat.



Scenario name: dynamic
Scenario description: varying duty cycl
◉ Dynamic power

Fig. 3.19 Floorplan and mapping of power traces to obtain temperature map. (**a**) Component-level floorplan. (**b**) Mapping of power traces to components

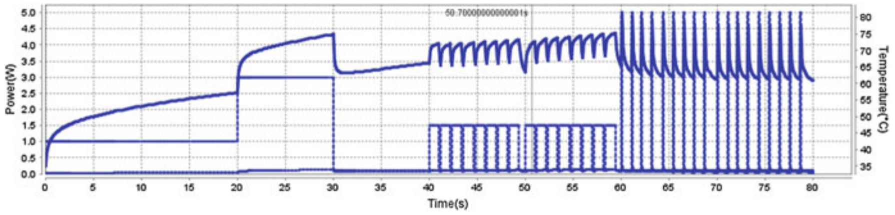


Fig. 3.20 Example: power $P(t)$ (*lower curve*) and temperature $\Theta(t)$ (*upper curve*) trace

3.6 Conclusion and Future Work

In this chapter, we presented our flow to consider power and temperature of a full chip in an executable system level model using a virtual platform. The presented mixed-criticality use-case described the integration of a mission- and a safety-critical application on the same MPSoC. The proposed hardware architecture of this MPSoC takes advantage of the heterogeneity and implements the mission-critical image processing on the high performance ARM dual core subsystem and the safety-critical real-time processing in the FPGA. The proposed power and temperature simulation has been applied to this heterogeneous system and was capable of providing feedback about the hidden interdependency of the mission- and the safety-critical applications due to thermal and electro-thermal coupling. With our extended virtual platform, software developers are capable to virtually integrate their applications on a single MPSoC and apply power and thermal management to run the system in safe operation under all possible environmental conditions.

The presented flow is still work in progress and the integration between the power and the temperature model is still under development. Currently the implementation of a virtual temperature sensor to provide direct feedback from the thermal

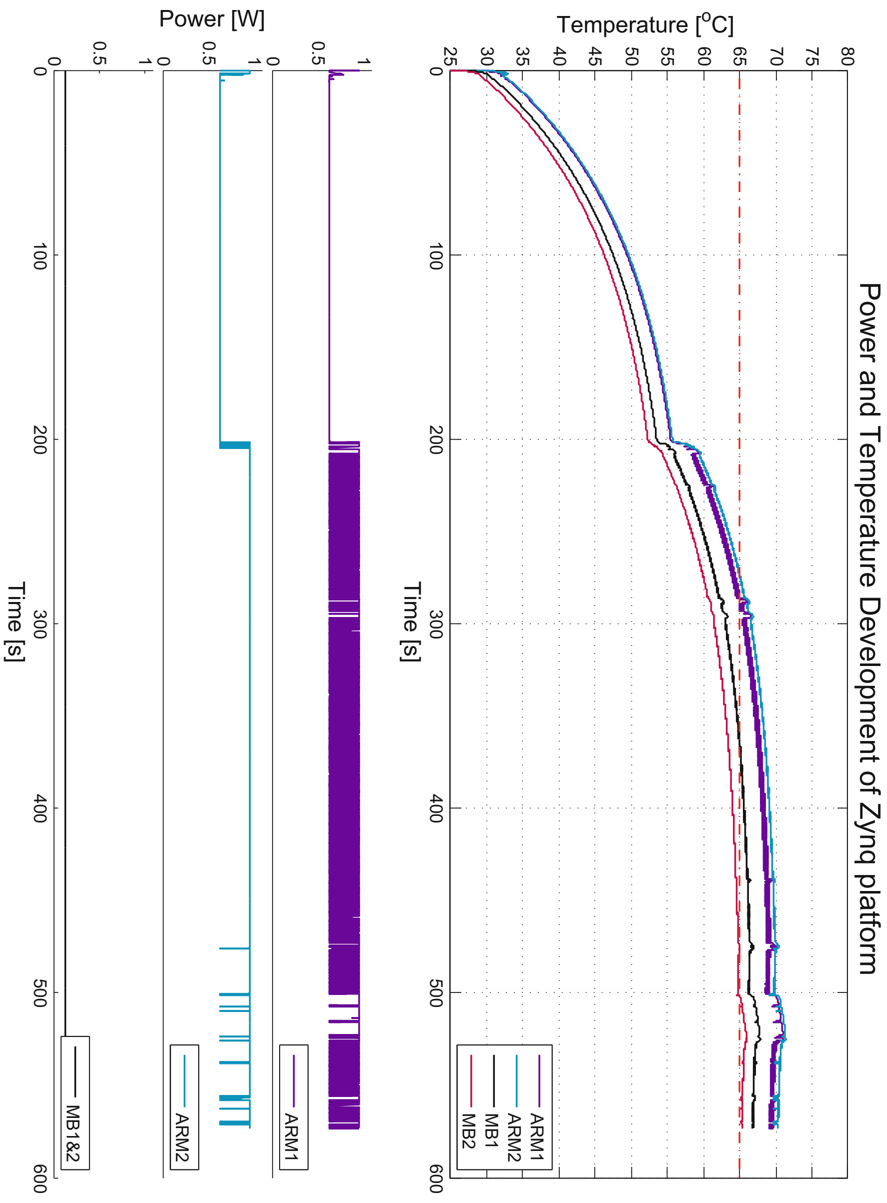


Fig. 3.21 Use-case power and temperature trace

simulation to running application (e.g., by dedicated HAL or OS services) is under development. After integration of the power and temperature model of the Zynq platform configuration, used in our case study, a validation against physical power and temperature measurements will be performed. Furthermore, explicit support for thermal interference analysis will be provided on top of the integrated model. The goal of this analysis is the online monitoring of thermal budgets per application and criticality level to drive the development of future resource and power management strategies for future mixed-criticality systems on a single chip.

Acknowledgements Many thanks to the CONTREX team members Ralph Görgen, Sven Rosinger, Sören Schreiner, Marco Feltes, and Martin Bornhold, my colleagues Henning Schlender and Malte Metzdorf, and the master students Jörn Bellersen, Martin Bornhold, Marco Braun, Henning Elbers, Thomas Nordlohne, Niklas May-Johann, Jenny Röbesaat, André Schaadt, Patrick Schmale, Steven Schmidt, Sebastian Vander Maelen, and Markus Wieghaus for their great work and enthusiasm on the multi-rotor use-case.

This work has been partially supported by the EU integrated project CONTREX (FP7-611146) and the ARTEMIS project EMC^2 (01-IS14002R) partially funded by the EC and the German Federal Ministry of Education and Research (BMBF).

References

1. J. Bellersen, M. Bornhold, M. Braun, H. Elbers, T. Nordlohne, N. May-Johann, J. Röbesaat, A. Schaadt, P. Schmale, S. Schmidt, S.V. Maelen, M. Wieghaus, Dokumentation Avionic Architecture. Tech. rep., Carl von Ossietzky Universität Oldenburg, Fakultät II - Informatik, Wirtschafts- und Rechtswissenschaften, Department für Informatik (2015)
2. A. Burns, R.I. Davis, Mixed Criticality Systems - A Review. http://www-users.cs.york.ac.uk/burns/review.pdf (2015)
3. FP7 EU project CONTREX (Design of embedded mixed-criticality CONTRol systems under consideration of EXtra-functional properties). http://contrex.offis.de
4. Docea Power: Aceplorer (2016), http://www.doceapower.com/index.php?option=com_content&view=article&id=1&Itemid=102
5. Docea Power: Thermal Profiler (2016), http://www.doceapower.com/index.php?option=com_content&view=article&id=237&Itemid=145
6. H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, D. Burger, Dark silicon and the end of multicore scaling. SIGARCH Comput. Archit. News **39**(3), 365–376 (2011). http://doi.acm.org/10.1145/2024723.2000108
7. K. Goossens, A. Azevedo, K. Chandrasekar, M.D. Gomony, S. Goossens, M. Koedam, Y. Li, D. Mirzoyan, A. Molnos, A.B. Nejad, A. Nelson, S. Sinha, Virtual execution platforms for mixed-time-criticality systems: The compsoc architecture and design flow. SIGBED Rev. **10**(3), 23–34 (2013). http://doi.acm.org/10.1145/2544350.2544353
8. K. Grüttner, P. Hartmann, T. Fandrey, K. Hylla, D. Lorenz, S. Stattelmann, B. Sander, O. Bringmann, W. Nebel, W. Rosenstiel, An esl timing amp; power estimation and simulation framework for heterogeneous socs. In *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, pp. 181–190 (July 2014)
9. P.A. Hartmann, K. Grüttner, W. Nebel, Advanced systemc tracing and analysis framework for extra-functional properties. In *Applied Reconfigurable Computing - 11th International Symposium, ARC 2015*, ed. by K. Sano, D. Soudris, M. Hübner, P.C. Diniz, Bochum, Germany, April 13-17, 2015. Proceedings. Lecture Notes in Computer Science, vol. 9040 (Springer, New York, 2015), pp. 141–152. http://dx.doi.org/10.1007/978-3-319-16214-0_12
10. D. Helms, K. Grüttner, R. Eilers, M. Metzdorf, K. Hylla, F. Poppen, W. Nebel, Considering variation and aging in a full chip design methodology at system level. In: *Proceedings of the 2014 Electronic System Level Synthesis Conference (ESLsyn)*, pp. 1–6 (May 2014)
11. HiSystems GmbH Germany. https://www.mikrocontroller.com/ (07 2015)

12. Imperas Software Limited: M*SDK - Advanced Multicore Software Development Kit (2016), http://www.imperas.com/msdk-advanced-multicore-software-development-kit
13. Imperas Software Limited: Open Virtual Platforms™(OVP™) (2016), http://www.ovpworld.org
14. N.S. Kim, T. Austin, D. Blaauw, T. Mudge, K. Flautner, J.S. Hu, M.J. Irwin, M. Kandemir, V. Narayanan, Leakage current: Moore's law meets static power. Computer **36**(12), 68–75 (2003). http://dx.doi.org/10.1109/MC.2003.1250885
15. S. Kumar Rethinagiri, O. Palomar, J. Arias Moreno, O. Unsal, A. Cristal, Vppet: Virtual platform power and energy estimation tool for heterogeneous mpsoc based fpga platforms. In: *24th International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS), 2014*, pp. 1–8 (Sept 2014)
16. A. Lanusse, Y. Tanguy, H. Espinoza, C. Mraidha, S. Gerard, P. Tessier, R. Schnekenburger, H. Dubois, F. Terrier, Papyrus uml: an open source toolset for mda. In *Proceedings ECMDA-FA '09: Model Driven Architecture - Foundations and Applications: 5th European Conference, ECMDA-FA 2009*, Enschede, The Netherlands, June 23–26, 2009, ed. by R.F. Paige, A. Hartman, A. Rensink. Lecture Notes in Computer Science, vol. 5562, pp. 1–4 (Springer, New York, 2009)
17. D. Lorenz, K. Grüttner, N. Bombieri, V. Guarnieri, S. Bocchio, From RTL IP to functional system-level models with extra-functional properties. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. CODES+ISSS '12 (ACM, New York, NY, USA, 2012), pp. 547–556. http://doi.acm.org/10.1145/2380445.2380529
18. D. Lorenz, K. Grüttner, W. Nebel, Data- and state-dependent power characterisation and simulation of black-box RTL IP components at system level. In: *17th Euromicro Conference on Digital Systems Design* (DSD 2014) (2014)
19. D. Lorenz, K. Grüttner, V. Ortland, Trace-based power state machine modelling. In: *Proceedings of the Forum on Specification and Design Languages* (FDL'2014) (2014)
20. D. Lorenz, P.A. Hartmann, K. Grüttner, W. Nebel, Non-invasive power simulation at system-level with SystemC. In: *Power and Timing Modeling, Optimization and Simulation - 22nd International Workshop* (PATMOS'2012). Lecture Notes in Computer Science (Springer, New York, 2012), pp. 21–31
21. W. Nebel, D. Helms, K. Grüttner, F. Oppenheimer, Closing the gap between technology and application needs (5 2013), edaWorkshop Panel
22. G. Nitsche, K. Grüttner, W. Nebel, Power contracts: A formal way towards power–closure?! In: *Proc. of the 23rd Intl. Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pp. 59–66 (September 2013)
23. G. Nitsche, K. Grüttner, W. Nebel, Towards satisfaction checking of Power Contracts in Uppaal. In *Proceedings of the 2014 Forum on Specification and Design Languages* (FDL), ed. by Chips, E.E.E., design Initiative, S. ECSI - European Electronic Chips and Systems design Initiative, München (Oct 2014)
24. Object Management Group (OMG): UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems (2011)
25. Object Management Group (OMG): OMG Systems Modeling Language (OMG SysML), Version 1.3 (2012), http://www.omg.org/spec/SysML/1.3/
26. Project Group Avionic Architecture (PGAA). https://www.uni-oldenburg.de/avionic-architecture/ (2015)
27. F. Rosa, L. Ost, R. Reis, G. Sassatelli, Instruction-driven timing cpu model for efficient embedded software development using ovp. In: *2013 IEEE 20th International Conference on Electronics, Circuits, and Systems* (ICECS), pp. 855–858 (Dec 2013)
28. S. Rosinger, M. Metzdorf, D. Helms, W. Nebel, Behavioral-level thermal- and aging-estimation flow. In: *Test Workshop (LATW), 2011 12th Latin American*, pp. 1–6 (March 2011)
29. H. Schlender, S. Schreiner, M. Metzdorf, K. Grüttner, W. Nebel, Teaching mixed-criticality: Multi-rotor flight control and payload processing on a single chip. In: *Proceedings of the 2015 Workshop on Embedded and Cyber-Physical Systems Education* (WESE) (10 2015)

30. S. Schreiner, K. Grüttner, S. Rosinger, A. Rettberg, Autonomous flight control meets custom payload processing: A mixed-critical avionics architecture approach for civilian uavs. In *Proceedings of the 2014 IEEE 17th International Symposium on Object/Component-Oriented Real-Time Distributed Computing*, ISORC '14 (IEEE Computer Society, Washington, DC, USA, 2014), pp. 348–357. http://dx.doi.org/10.1109/ISORC.2014.28
31. Trenz Electronic GmbH: TE0720 Series (Z-7020), http://www.trenz-electronic.de/de/produkte/fpga-boards/trenz-electronic-te0720-zynq.html
32. S. Trujillo, R. Obermaisser, K. Grüttner, F.J. Cazorla, J. Perez, European Project Cluster on Mixed-Criticality Systems. In *3PMCES Workshop (Performance, Power and Predictability of Many-Core Embedded Systems) at DATE'14*. Electronic Chips & Systems Design Initiative (ECSI) (2014)
33. Xilinx Inc.: Zynq™-7000 SoC Extensible Virtual Platform, http://www.xilinx.com/products/zynq-7000/extensible-virtual-platform.htm
34. Xilinx Inc.: Xilinx Power Estimator (XPE) (2016), http://www.xilinx.com/products/design_tools/logic_design/xpe.htm
35. Xilinx Inc. Zynq-7000 All Programmable SoC, http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html (07 2015)

Chapter 4
Throughput-Driven Parallel Embedded Software Synthesis from Synchronous Dataflow Models: Caveats and Remedies

Matin Hashemi, Kamyar Mirzazad Barijough, and Soheil Ghiasi

4.1 Introduction

The model-based design methodology advocates separation of application specification from target implementation, and representation of application behavior using formal models of computation [36, 37]. Such models enable one to develop or to utilize various analysis, optimization, and synthesis techniques for either exploration of implementation space or generation of efficient implementations. While this approach has unquestionable benefits, we argue that in certain situations *complete* separation of specification from details of the target implementation obscures key pieces of information that are essential for accurate characterization of the design space.

We study specific manifestations of this general observation in the context of embedded streaming applications modeled as synchronous dataflow (SDF) graphs that are to be implemented on multiprocessor system on chip (MPSoC) platforms. Streaming applications appear in many disciplines such as networking, signal processing, security, and multimedia. They are characterized by the requirement to process a virtually infinite sequence of data items typically under throughput constraints [24, 34, 46]. Therefore, we focus on streaming throughput as the key quality metric in our discussions.

In particular, we study two problems in the *model analysis* and *model synthesis* space. First, in Sect. 4.2, we discuss buffer-throughput tradeoff analysis for an arbitrary SDF model. We show that state of the art throughput analysis techniques

M. Hashemi (✉) • K.M. Barijough
Sharif University of Technology, Tehran, Iran
e-mail: matin@sharif.edu; kammirzazad@ee.sharif.edu

S. Ghiasi
University of California, Davis, CA, USA
e-mail: ghiasi@ucdavis.edu

A. Molnos, C. Fabre (eds.), *Model-Implementation Fidelity in Cyber Physical System Design*, DOI 10.1007/978-3-319-47307-9_4

that operate solely based on SDF graph operational semantics yield overly pessimistic throughput estimations. We propose a set of model transformations that embed some platform-inspired information in the SDF model. We demonstrate that the limited amount of platform awareness significantly improves model-based throughput analysis [3].

In Sect. 4.3, we focus on the problem of scaling streaming throughput for a given SDF model, as the software synthesis engine targets different platforms. We demonstrate that conventional SDF graph synthesis schemes cannot scale throughput well, as the number of processor cores in the platform increases. We argue that fully decoupled specification of the model from the target implementation is a key obstacle, and thus, we outline a platform-inspired malleable dataflow graph specification model that addresses the throughput scaling issue. Extensive empirical evaluations validate the efficacy of our approach [23].

4.2 Streaming Throughput Analysis

4.2.1 Overview

SDF applications are represented as a set of concurrent tasks that communicate by sending and receiving messages (tokens) via point-to-point FIFO buffers [28, 46]. The rates at which tasks produce and consume messages are constant and known at compile time. SDF operational semantics specifies consumption of all input messages to a task upon start of its execution, and production of all its output messages upon completion of its execution. An implementation oblivious analysis technique would have to follow model execution according to the operational semantics. In actual implementations, however, not all messages of a task are consumed or produced at exactly the same time. Presence of limited information or mild assumptions about the nature of target implementation would increase the timing resolution during model execution. For example, if one assumes that tasks are going to be implemented as software modules running on parallel processors, a sequential order would have to be imposed on the production and consumption of messages. This breaks the pessimistic simultaneous message production and consumption that is dictated by the SDF operational semantics, and potentially leads to more accurate analysis.

We utilize the state of the art implementation oblivious throughput analysis technique developed by Stuijk et al. [41], and argue that its throughput estimation is overly pessimistic. We propose transformations to the application SDF model to capture the sequential nature of message production and consumption by software, and to rigorously embed implementation awareness into the model. Subsequently, we leverage the method of Stuijk et al. for throughput analysis of the transformed SDF model. The additional information that we expose to the throughput analysis algorithm are quite limited in nature: merely sequential order between production and consumption of messages, which is implied by the assumption of implementation as software. As such, the analysis is not tied to the *details* of target MPSoC execution platform, and would complement, rather than contradict with, the model-based design paradigm.

4.2.2 Preliminaries

4.2.2.1 SDF Model

SDF graph is a directed graph $G(V, E)$, where vertex $v \in V$ represents an actor, and edge $uv \in E$ represents a point-to-point FIFO *channel* from actor u to v. Actors communicate by sending/receiving data items, called *tokens*, via the channels. Actor v is a tuple $(\mathrm{In}, \mathrm{Out}, F, \varepsilon)$ and channel uv is a tuple (u, v, r_p, r_c). $\mathrm{In}(v) \subset E$ and $\mathrm{Out}(v) \subset E$ are input and output channels of v, $F(v)$ is the actor's transformation function, and $\varepsilon(v)$ is its execution time, i.e., the average time actor v takes to perform the transformation function in an implied implementation (Fig. 4.1a). For a channel $uv \in E$, the number of tokens produced by u for channel uv, on every firing of u, is called the production rate of uv and is denoted by $r_p(uv)$. Consumption rate $r_c(uv)$ is defined similarly. Data rates are constant and actor execution is meant to continue infinitely [27, 41].

Execution (Firing) Condition: Actor v can execute, also known as fire, at time t, if and only if (I) previous firings of v have completed,[1] and (II) enough tokens are available on all of its input channels, that is $\forall uv \in \mathrm{In}(v) : \gamma(uv, t) \geq r_c(uv)$, where $\gamma(uv, t)$ quantifies the number of tokens stored in uv at time t.

SDF Operational Semantics: Upon scheduling of actor v for execution, it simultaneously consumes $r_c(uv)$ tokens from all of its input channels $uv \in \mathrm{In}(v)$, then carries out its computation in $\varepsilon(v)$ time units, and finally it simultaneously produces $r_p(vw)$ tokens on all of its output channels $vw \in \mathrm{Out}(v)$. Figure 4.1a shows an example in which $\varepsilon(b) = 300$, $r_c(ab) = 50$, and $r_p(bc) = 10$. Thus, upon availability of at least 50 tokens on ab, actor b can fire. In every firing of b, 50 tokens are simultaneously consumed from ab, then the computation of actor b is carried out in 300 time units, and finally ten tokens are simultaneously written to bc.

4.2.2.2 Target Platform Model

We target MPSoC platforms whose abstract model for SDF execution can be viewed as a distributed-memory message-passing system with point-to-point interprocessor FIFO buffers (Fig. 4.1b). This abstract view is directly implemented in some

[1]Auto-concurrency, i.e., multiple concurrent firings of an actor, is not allowed in our discussion.

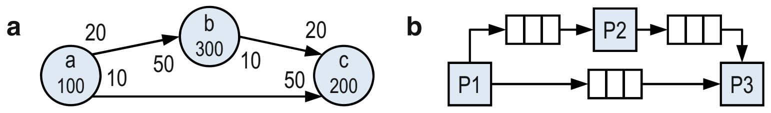


Fig. 4.1 (**a**) Example SDF graph (actors and channels are annotated with execution times and data rates, respectively.) (**b**) An implied implementation of self-timed execution

platforms such as AsAP [49] and TILE64 static network [5]. Some other platforms implement the abstract view via circular arrays that are allocated in the shared memory, using proper producer–consumer synchronization schemes. Regardless and for sake of our discussion, the platform can be abstractly viewed as a multiprocessor with a FIFO interconnection network.

We focus on self-timed execution, which implicitly assumes allocation of dedicated execution resources to every actor (Fig. 4.1b). Under self-timed execution, an actor fires as soon as its firing conditions are satisfied [41]. In many cases, an embedded application is developed on an MPSoC target by splitting the application into many actors, and assigning each actor to its dedicated core (e.g., 1080p H.264 encoder on AsAP [49]). Otherwise, the collection of actors allocated to the same processor under static schedule can be viewed as a coarse-grain actor in an upscaled version of the graph that conforms to our model.

4.2.2.3 Buffer-Throughput Tradeoff

Throughput[2] is one of the most important quality metrics in streaming applications. A number of factors, such as actor execution times, actor allocation and scheduling on processor cores, interprocessor buffer sizes, SDF graph structure, and SDF graph cycles, impact steady-state throughput [16, 20–22, 24, 41]. In practice, the FIFO channels must be implemented with finite buffering capacity, which may limit the throughput, and hence, there is a tradeoff between interprocessor buffer sizes and application throughput [41].

Throughput: Throughput of an actor v is defined as the average number of v firings per unit time [16], i.e., $\tau(v) = \lim_{T\to\infty} \frac{1}{T} \times \big(\#\text{ of } v \text{ firingsfrom } t = 0 \text{ to } t = T\big)$. Since SDF data rates are constant, in the steady-state, the number of times different actors fire are a constant factor of one another. Hence, normalized throughput, which decouples the choice of actor from SDF throughput, is defined as $\tau = \tau(v) \div q(v)$ for an arbitrary actor $v \in V$, where, $q(v)$ is the number of times v fires in one iteration of the simplest periodic schedule [16, 27]. In our example, $q(a, b, c) = (5, 2, 1)$.

[2]Here, we use the terms "steady-state throughput" and "throughput" interchangeably.

Buffer Size: Buffer size $\beta(uv)$ is defined as capacity of the interprocessor FIFO buffer which implements channel $uv \in E$. In other words, $\beta(uv)$ is the maximum number of tokens that channel uv can hold at any time during execution. Formally, $\gamma(uv, t) \leq \beta(uv)$. Total buffer size is defined as $|\beta| = \sum_{\forall uv \in E} \beta(uv)$.

4.2.3 Inaccuracy in SDF-Based Throughput Analysis

4.2.3.1 Throughput Analysis Based on SDF Operational Semantics

According to SDF operational semantics, after actor u fires and completes its computation, at least $r_p(uv)$ empty spaces are required on every output channel $uv \in \text{Out}(u)$ in order to write tokens produced by u. Otherwise, since sufficient space is not available, u is stalled at the end of its firing. The actor will resume execution to complete its previously stalled firing only after enough space becomes available.

Stall and Resume Conditions: Under self-timed execution assumption, a running actor $u \in V$ fired at time t_1 stalls at time $t_2 > t_1$ if and only if $\exists uv \in \text{Out}(u) : \beta(uv) - \gamma(uv, t_2) < r_p(uv)$ and resumes operation at a time $t_3 > t_2$ if and only if $\forall uv \in \text{Out}(u) : \beta(uv) - \gamma(uv, t_3) \geq r_p(uv)$.

Throughput is degraded if actors stall due to unavailable space. For a given set of buffer sizes β, throughput can be obtained by considering the firing, stall, and resume conditions. Stuijk et al. developed a Pareto point exploration algorithm to find throughput vs. total buffer size of an SDF graph [41]. The algorithm works by executing the SDF graph, for a judiciously selected subset of buffer size allocations, while maintaining the state of actors and channels. Each step of application execution is modeled as a transition in the augmented state space of actors and channels. When a state is revisited for the first time, the execution arrives its steady-state, as a cycle in the state space is formed. Subsequently, throughput $\tau(v)$ is calculated as the number of v firings during the cycle, divided by the amount of time lapsed in the cycle. The above procedure is repeated for different buffer sizes in order to evaluate all Pareto points [41]. We later utilize this algorithm in our experimentation in Sect. 4.2.5.

Figure 4.2 demonstrates throughput calculation for our running example of Fig. 4.1 when $\beta(ab, ac, bc) = (60, 50, 20)$. At time $t = 1100$, the progress and capacities of all actors and channels are equal to those of time $t = 300$. Thus, the steady-state is reached, and $\tau(b) = \frac{2}{1100-300}$ and $\tau = \frac{\tau(b)}{q(b)} = \frac{1}{800}$. If the buffer size of channel ac is increased from 50 to 70, throughput improves from $1/800$ to $1/600$, because channel ac becomes full 200 time units later, and actor a stalls for 200 fewer time units.

Deadlock: When at least one stalled actor never resumes operation, then deadlock happens, in which case, overall throughput τ becomes zero. By analyzing SDF operational semantics, Ade et al. [1] proved the following theorem regarding deadlocks:

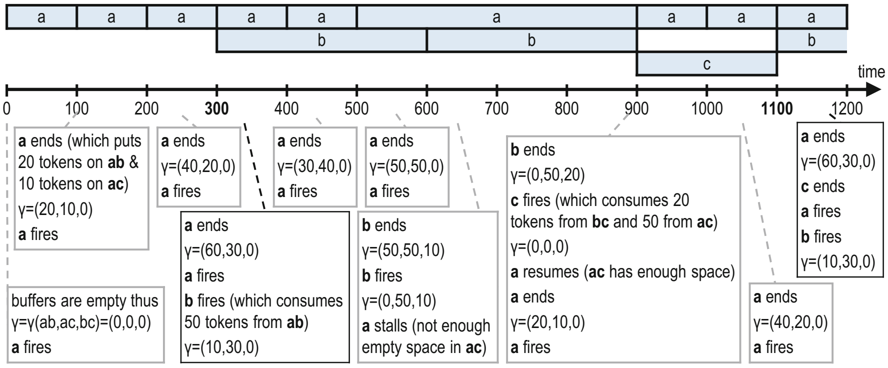

a a a a a a a a a
b b b
c
time
0 100 200 300 400 500 600 700 800 900 1000 1100 1200
buffers are empty thus γ=γ(ab,ac,bc)=(0,0,0) a fires
a ends (which puts 20 tokens on ab & 10 tokens on ac) γ=(20,10,0) a fires
a ends γ=(40,20,0) a fires
a ends γ=(60,30,0) a fires b fires (which consumes 50 tokens from ab) γ=(10,30,0)
a ends γ=(30,40,0) a fires
a ends γ=(50,50,0) a fires
b ends γ=(50,50,10) b fires γ=(0,50,10) a stalls (not enough empty space in ac)
b ends γ=(0,50,20) c fires (which consumes 20 tokens from bc and 50 from ac) γ=(0,0,0) a resumes (ac has enough space) a ends γ=(20,10,0) a fires
a ends γ=(40,20,0) a fires
a ends γ=(60,30,0) c ends a fires b fires γ=(10,30,0)

Fig. 4.2 Throughput analysis based on SDF operational semantics when $\beta(ab, ac, bc) = (60, 50, 20)$. At $t = 1100$, states of actors and channels are the same as $t = 300$

(a)

```
// task 'a' on P1
token ab[20];
token ac[10];

while(){
 a(ab,ac);
 write(ab,20,P2);
 write(ac,10,P3);
}
```

```
// task 'b' on P2
token ab[50];
token bc[10];

while(){
 read(ab,50,P1);
 b(ab,bc);
 write(bc,10,P3);
}
```

```
// task 'c' on P3
token bc[20];
token ac[50];

while(){
 read(bc,20,P2);
 read(ac,50,P1);
 c(bc,ac);
}
```

(b)

```
void write (token* x, int n, int dst){
 for i=[0,n)
  for j=[0,s)
   writePacket(x[i],j,dst);
}
```

```
void read (token* x, int n, int src){
 for i=[0,n)
  for j=[0,s)
   readPacket(x[i],j,src);
}
```

Fig. 4.3 Abstract view of (**a**) software implementation, and (**b**) communication APIs

Theorem 1. *If there exists a channel $uv \in E$ with buffer size $\beta(uv)$ less than*

$$r_p(uv) + r_c(uv) - gcd(r_p(uv), r_c(uv))$$

then deadlock happens between actors u and v. Here, gcd refers to greatest common divisor operation. We denote the above equation with $\beta_{\min}(uv)$.

In our example, $\beta_{\min}(ab, ac, bc) = (60, 50, 20)$, which means, if interprocessor buffer sizes $\beta(ab) < 60$ or $\beta(ac) < 50$ or $\beta(bc) < 20$, then deadlock happens. Note that the theorem does not state if deadlock happens when "for all" channels $uv \in E$, $\beta(uv) \geq \beta_{\min}(uv)$. In such a case, more thorough deadlock analysis is required [50].

4.2.3.2 Abstract View of Implementation

Figure 4.3a demonstrates our abstract view of embedded software that implements the SDF application on an MPSoC. First, the required tokens are read from input FIFO buffers, next the actor's specific computation is executed, and finally, the generated data is written to output buffers. This sequence is repeated indefinitely. Let us define "task" as "implementation of actor" according to this abstract view.

Figure 4.3b shows the typical implementation of communication API calls. The SDF model allows tokens of arbitrary size, hence, one may define a large block of data, such as a video frame, as a single token. However, interconnect networks have limited bandwidth and they are not necessarily capable of transferring one token at a time (e.g., one video frame takes multiple clock cycles). In practice, each token may need to be split into $s = \lceil \frac{\text{sizeof(token)}}{\text{sizeof(packet)}} \rceil$ packets, which have to be transferred sequentially as shown in the inner loop of Fig. 4.3b. The outer loop repeats this process for every token in the array. For brevity, we assume $s = 1$ in the rest of this paper. Our approach, however, is readily extensible to other packet sizes.

Note that this abstract view refers to very general implementation guidelines, rather than a specific platform or software coding style. Therefore, many different concrete implementations conform to this abstract view, albeit with different parameters. For example, many interprocessor API calls which appear atomic to the programmers are implemented by splitting large data into smaller pieces and transferring them sequentially. As another example, inter-processor token transfer through blocking API calls to DMA hardware also falls in this category because tokens are still transferred sequentially but by the DMA hardware rather than the processors. However, non-blocking token transfer is not covered in this abstract view of implementation. As another example, in software implementations conceptually concurrent token transfer would have to be implemented in some sequential order.

In practice, when SDF graph is implemented in a form that conforms to our abstract implementation, the simultaneity in reading and writing tokens at arbitrary rates is not faithfully implemented. The sequential nature of instruction execution on single-issue processor cores implies that a task can write (read) only one token to (from) only one channel at a time. This additional information about implementations leads to an operation that is quite different from the pure SDF model, in which actors write to (read from) all channels simultaneously at specified rates.

As shown in Fig. 4.2, analysis based on SDF model concluded that throughput for interprocessor buffer size $\beta(ab, ac, bc) = (60, 50, 20)$ is $\tau = \frac{1}{800}$. Actor c waits for data from b and upon availability of sufficient number of tokens produced by b, actor c fires and immediately consumes all of them.

The implementation, however, behaves differently by allowing tasks to only read and write one token at a time (Fig. 4.3). Task c (processor P3) stalls when it tries to read for the first time, since there is no token available on channel bc. Once task b (processor P2) places the first token on bc, the stalled `readPacket` function in c resumes execution and reads that token. In this setting, therefore, $\beta(bc) = 1$ would be sufficient to achieve the same throughput as shown in Fig. 4.2. This amounts to a substantial $20\times$ reduction in size of the interprocessor buffer bc without any throughput degradation. Interestingly, the analysis based on Theorem 1 and SDF operational semantics (Sect. 4.2.3.1) resulted in $\beta_{\min}(bc) = 20$, which implies that a buffer size of $\beta(bc) = 1$ should result in deadlock and hence a throughput of zero. However, in practice it does not cause deadlock or even decrease the throughput for the target implementation.

As highlighted in the above example, the analysis solely based on SDF operational semantics leads to much larger interprocessor buffers than actually required in practical implementations to achieve a certain level of throughput, or in other words, leads to much smaller throughput than actually happens in practical implementations with certain interprocessor buffer sizes.

4.2.4 *Proposed Solution: Implementation Aware Throughput Analysis*

We propose taking into account a key piece of information about target MPSoC implementations by which throughput analysis would become more accurate. In our discussion, we adopt the above abstract view of implementation for an application modeled as an SDF graph.

We take a two step approach to bring implementation awareness into throughput analysis. First, we transform the original SDF graph G by embedding implementation-dictated sequential data production and consumption into the graph (Fig. 4.6b). Clearly, the transformation must preserve the function and other relevant aspects of the original application. Subsequently, the transformed SDF graph G' is analyzed by leveraging an implementation oblivious technique, described earlier in Sect. 4.2.2.1.

Based on the abstract view of implementation, tasks can read (write) only one token at a time (property I), and from (to) only one channel at a time (property II). Our proposed SDF graph transformation models these two properties by adding *virtual* actors and channels to the SDF graph. Specifically, property I is modeled by adding virtual *reader* and *writer* actors, and property II is captured by adding virtual *sync* actors to the SDF graph.

4.2.4.1 Reader and Writer Actors

For every channel $uv \in E$, a virtual writer actor W is added at the output of actor u, and a virtual reader actor R is added at the input of actor v, such that the output of W feeds data into the input of R (Fig. 4.4a). All reader and writer actors have identity data transformation functionality and thus, do not alter the data.

Reader and writer actors have production and consumption rates of 1. Thus, for every firing of u, W has to fire $r_p(uv)$ times sequentially to consume the tokens produced by u one at a time. Recall that auto-concurrency is disallowed in our discussion. Similarly, for every $r_c(uv)$ firings of R, actor v fires once. Buffer sizes for channels uW and Rv are set to $r_p(uv)$ and $r_c(uv)$, respectively. Buffer size of channel uv in the original graph determines buffer size of WR in the transformed graph (Fig. 4.4a).

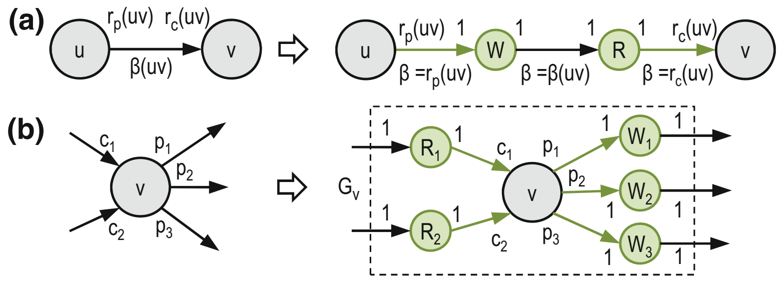


Fig. 4.4 (**a**) Writer and reader actors for channel $uv \in E$. Virtual actors and channels are shown in *green*. (**b**) The transformed subgraph G_v for an actor v with two incoming and three outgoing channels

Writer actor W models behavior of the `writePacket` function call (Fig. 4.3b). $r_p(uv)$ firings of W, which produce $r_p(uv)$ tokens, model the loop, and iterative calls to `writePacket` function in the `write` API call in execution of task u. Intuitively, virtual channel uW models the local processor memory that temporarily stores the output tokens of u (e.g., `token ab[20]` in task a in Fig. 4.3a). Similarly, actor R models the `readPacket` call, and channel Rv models the local memory that temporarily stores the input tokens of a task v (e.g., `token ab[50]` in task b in Fig. 4.3a).

As a result of the above transformation, every actor $v \in V$ is transformed into a subgraph G_v (Fig. 4.4b). Let $|\mathrm{In}(v)|$ and $|\mathrm{Out}(v)|$ denote the number of input and output channels of v. Let c_i for $i \in [\,1, |\mathrm{In}(v)|\,]$ denote the consumption rates for input channels of v, and let p_j for $j \in [\,1, |\mathrm{Out}(v)|\,]$ denote the production rates for output channels of v. Subgraph G_v has $|\mathrm{In}(v)|$ reader actors $R_1, R_2, \ldots R_{|\mathrm{In}(v)|}$, and $|\mathrm{Out}(v)|$ writer actors $W_1, W_2, \ldots W_{|\mathrm{Out}(v)|}$. Data production ($r_p$) and consumption rates (r_c), and buffer sizes (β) of virtual channels in G_v are set as:

virtual channel	r_p	r_c	β
$R_i v$	1	c_i	c_i
vW_j	p_j	1	p_j

A firing of actor v in G corresponds to the following sequence of events in subgraph G_v in the transformed graph G'. Reader actor R_i fires c_i times. As a result, it reads c_i tokens from the corresponding input channel of G_v and writes them to virtual channel $R_i v$. At this point, actor v fires once and consumes all of the input tokens and produces p_j tokens on virtual channels vW_j. Next, virtual actor W_j fires p_j times, and copies the tokens to the corresponding output channel of G_v. Note that input channels of G_v have consumption rates of 1 because they are connected to reader actors. Similarly, output channels of G_v have production rates of 1. Thus, subgraph G_v models the execution of task v based on the implementation view discussed in Sect. 4.2.3.2.

Theorem 2. *Addition of reader and writer actors preserves SDF functionality.*

Proof. SDF functionality is independent of task execution order (scheduling), and merely depends on the value and order of data tokens in channels [8]. Both reader and writer actors have the identity transfer function and do not alter data. Moreover, they preserve the order of data tokens delivered from the producer to the consumer. Therefore, the end to end functionality of the SDF graph remains intact. □

4.2.4.2 Sync Actors

In subgraph G_v developed above, reader actors, writer actors, and actor v can potentially fire simultaneously. In order to correctly model the sequential nature of data consumption, computation, and data production based on the abstract implementation view, we need to eliminate the simultaneity. Our approach is to add a number of virtual sync actors to every subgraph G_v in order to enforce the following sequential ordering on the execution of actors:

$$R_1, R_2, \ldots R_{|\mathrm{In}(v)|}, v, W_1, W_2, \ldots W_{|\mathrm{Out}(v)|}$$

This sequential ordering conforms to the implementation of task v, where first the `read` API calls, next the computation of actor v, and finally the `write` API calls are executed on the processing core (Fig. 4.3a).

Specifically, to enforce the above ordering in G_v, we add virtual sync actors $S^R_{i,i+1}$ between R_i and R_{i+1}, and virtual sync actors $S^W_{j,j+1}$ between W_j and W_{j+1} (e.g., S_1, S_2, and S_3 in Fig. 4.5a), and set data production (r_p) and consumption (r_c) rates, and buffer size (β) of the newly added virtual channels (marked blue in the figure) as follows:

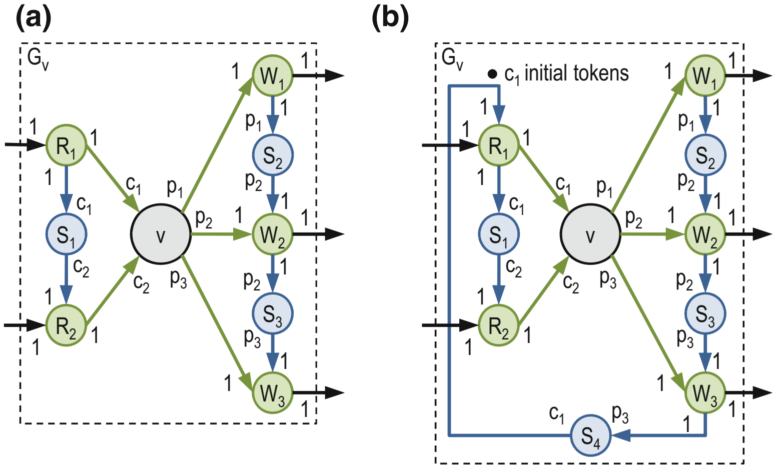


Fig. 4.5 (**a**) Sync actors S_1, S_2, and S_3 enforce the sequential order $R_1, R_2, v, W_1, W_2, W_3$ in subgraph G_v of Fig. 4.4b. The newly added virtual actors and channels are shown in *blue*. (**b**) Sync actor S_4 prohibits auto-concurrency

virtual channel	r_p	r_c	β	virtual channel	r_p	r_c	β
$R_i S^R_{i,i+1}$	1	c_i	c_i	$W_j S^W_{j,j+1}$	1	p_j	p_j
$S^R_{i,i+1} R_{i+1}$	c_{i+1}	1	c_{i+1}	$S^W_{j,j+1} W_{j+1}$	p_{j+1}	1	p_{j+1}

The parameters are carefully selected such that upon c_i firings of R_i, $S^R_{i,i+1}$ fires once, and then R_{i+1} can fire c_{i+1} times. Similarly, upon p_j firings of W_j, $S^W_{j,j+1}$ fires once, and then W_{j+1} can fire p_{j+1} times. By creating appropriate dependencies, the construction ensures that the desired ordering is enforced.

Lastly, we add a sync actor between $W_{|\mathrm{Out}(v)|}$ and R_1 (e.g., S_4 in Fig. 4.5b). This creates a cycle in G_v and prohibits concurrent execution of a reader actor and a writer actor. Specifically, it stops R_1 from firing until $W_{|\mathrm{Out}(v)|}$ fires $p_{|\mathrm{Out}(v)|}$ times. Note that c_1 initial tokens are required on this cycle in order to avoid deadlock, since R_1 fires c_1 times for every firing of v.

Sync actors have no effect on the transfer function of reader/writer actors. In particular, the reader and writer actors continue to copy application data (black and green channels in Fig. 4.5b), and do not mix up the data with dependency channels of the sync actors (blue channels in Fig. 4.5b). It follows that the transformed subgraph G_v in G correctly models the execution of task v according to the abstract view discussed in Sect. 4.2.3.2.

Theorem 3. *Addition of sync actors preserves the original SDF functionality.*

Proof. By construction, read and write actors do not mix up application data tokens (green channels) with synchronization tokens (blue channels). The application functionality merely depends on the data values and their ordering on green channels, which is isolated from sync actors. As such, sync actors have no impact on original SDF functionality. □

4.2.4.3 Properties

Since sync actors are added to only enforce a sequential order among read and write operations, they must not have any impact on the total execution time of G_v. We conservatively assume that the information regarding platform-dependent latency of read and write operations are unavailable. Hence, the execution times of `read` and `write` API calls and the data transformation computation of a task are viewed to be inseparable. To capture this in subgraph G_v, we set the execution times of reader and writer actors to zero ($\varepsilon = 0$), and assign the entire execution time of the original actor to v. In case of access to specific parameters of the target architecture, one could improve the model fidelity by separating the latency of read and write operations from data transformation computation, and assigning more accurate execution times to actors in G_v.

Theorem 4. *Assume that the set of actors in G_v in the transformed graph and actor v in the graph G start execution at the same time and from the same buffer state in G. In that case, if G_v stalls during execution, the corresponding actor v in the graph G must also stall.*

Proof. Stalling execution occurs because at least one channel does not have enough capacity to receive the produced tokens at that point in time. By construction, stalling G_v implies that at least one of output channels of the write actors does not have sufficient capacity during execution of G_v, as channels internal to G_v are allocated sufficient capacity to execute G_v. Since production rate of write actors is one (the smallest possible rate), at least one of the output channels of G_v must be entirely full. Execution of G_v takes exactly the same time as execution of v, and thus, the corresponding output channel of v in G is also full during execution of v, which would stall the execution of v. □

The following theorem articulates the pessimistic nature of implementation oblivious analysis.

Theorem 5. *Given an SDF graph G and a set of buffer size choices β for channels in G, throughput of transformed graph G' is not less than G.*

Proof. Assume that the theorem does not hold. Then, there must be a time t at which, *for the first time*, the execution of an actor v in G starts earlier than the corresponding execution of the graph G_v in the transformed graph. Execution of v at t implies that both conditions I and II discussed in Sect. 4.2.2.1 are satisfied at t.

Condition I indicates that no other firing of v is stalled. Based on theorem 4 there cannot be a stalled version of G_v, since t is the first point in time at which G supposedly runs ahead of the transformed graph. Hence, condition I is also satisfied for G_v. Condition II indicates that all of the input channels of v have sufficient number of tokens available at time t. Input channels of v form input channels of the read actors in G_v. Moreover, the consumption rate of read actors is one, which is the smallest possible valid rate. Thus, there must be sufficient number of tokens for read actors of G_v at time t, and its execution should not be stalled. The contradiction proves the theorem. □

As one would expect, the two original and transformed graphs should yield the same throughput if buffer capacity constraint is relaxed.

Theorem 6. *The maximum throughput of G and G', which is obtained when all channels of G have infinite buffer size, are equal.*

Proof. If buffer sizes are sufficiently large, throughput would be limited by the slowest actor or the iteration bound of the corresponding "homogeneous" SDF graph (HSDF). The iteration bound of a HSDF graph is equal to its maximum cycle mean, which is defined as the cycle latency divided by the number of initial tokens in the cycle [16, 35].

The transformation only adds actors with zero execution time to the graph, and hence, the slowest actor would have the same execution time in both graphs. The transformation creates cycles within the subgraph G_v, however, all such cycles have latency of $\varepsilon(v)$. There is at least one initial token in all cycles inside G_v, as the feedback edge from the last writer actor to the first reader actor must be part of the cycle. Thus, the cycle mean for cycles that are created inside G_v is not more than $\varepsilon(v)$, which would not limit the throughput. Finally, for cycles in the transformed graph that are not inside G_v, neither the cycle latency nor the number of initial tokens in the cycle are changed under the transformation, and hence, the two graphs will have the same limit throughput. □

4.2.5 *Empirical Evaluation*

4.2.5.1 Setup and Benchmark Applications

To evaluate the proposed technique we employ StreamIt benchmark applications. StreamIt is a programming language and compiler for stream programs [46]. For every benchmark application, we execute StreamIt compiler (Fig. 4.6, top left) and then extract SDF graph topology, data rates (r_p and r_c), and estimates of actor execution time (ε).

Actor execution times are estimated by the StreamIt compiler based on rough mapping between high-level StreamIt language constructs and typical processor instruction sets. Original cycle per instruction (CPI) estimates of StreamIt compiler

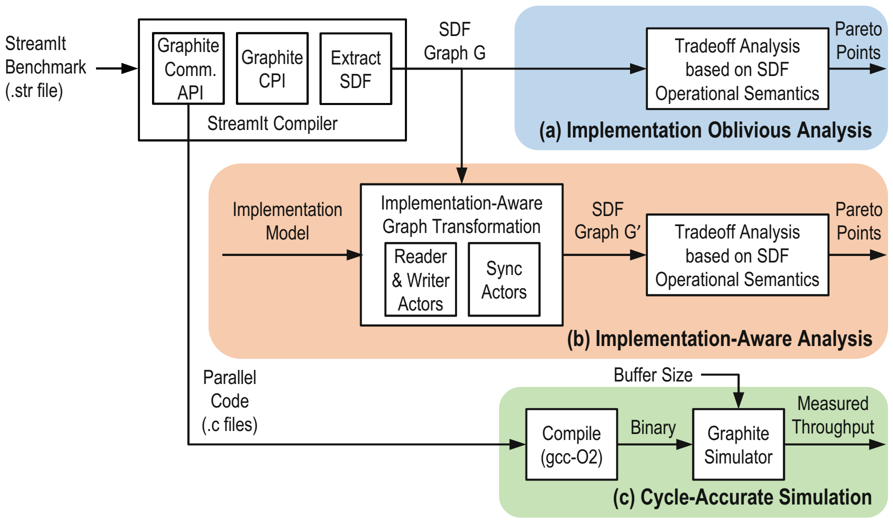

StreamIt Benchmark (.str file)
Graphite Comm. API
Graphite CPI
Extract SDF
StreamIt Compiler
SDF Graph G
Tradeoff Analysis based on SDF Operational Semantics
Pareto Points
(a) Implementation Oblivious Analysis
Implementation Model
Implementation-Aware Graph Transformation
Reader & Writer Actors
Sync Actors
SDF Graph G′
Tradeoff Analysis based on SDF Operational Semantics
Pareto Points
(b) Implementation-Aware Analysis
Parallel Code (.c files)
Buffer Size
Compile (gcc-O2)
Binary
Graphite Simulator
Measured Throughput
(c) Cycle-Accurate Simulation

Fig. 4.6 Experimentation flow: (**a**) Baseline implementation oblivious buffer-throughput tradeoff analysis based on SDF operational semantics. (**b**) Proposed implementation aware tradeoff analysis. (**c**) Cycle-accurate simulation of the compiled binary code

are based on the RAW processor. We have modified StreamIt source code such that its CPI estimates match Graphite processor model [18]. Graphite is a cycle-accurate MPSoC simulator, and is used as the target platform in our experimentation. The above procedure yields a series of SDF graphs which are used as benchmarks in our experimentation. We have released the generated SDF graphs along with details of the above procedure on the web [6].

Note that in our experimentation we evaluate the throughput for a range of buffer sizes in the target implementation and present a Pareto chart for the tradeoff between throughput and total buffer size.

4.2.5.2 Implementation Aware vs. Implementation Oblivious Analysis

The proposed implementation aware analysis involves two steps (Fig. 4.6b). First, we apply the proposed graph transformation discussed in Sect. 4.2.4 and transform the SDF graph G into another SDF graph G'. The transformation is based on our abstract view of target implementation as discussed in Sect. 4.2.3.2, which includes very limited information on the target (sequentially ordered read/write operations) into SDF graph G'.

Next, we perform buffer-throughput tradeoff analysis on G' based on SDF operational semantics, as discussed in Sect. 4.2.2.1. Here, we utilize SDF3 [39, 41], which implements the tradeoff analysis algorithm explained in Sect. 4.2.2.1. We modified SDF3 to force it to ignore the virtual channels introduced by the transformation, while exploring the search space. Buffer size of the virtual channels are also omitted from the reported total buffer size because virtual channels do not exist in the implementation and do not take any real space. The analysis yields a set of Pareto optimal points between total interprocessor buffer size, $|\beta|$, and corresponding overall throughput, τ. To compare the proposed approach against an established standard, we also perform implementation oblivious analysis directly on graph G (Fig. 4.6a).

Figure 4.7 shows the result of tradeoff analysis for both the proposed implementation aware and the baseline implementation oblivious techniques. The experimental results show that for all benchmarks the implementation aware tradeoff analysis yields much smaller buffer sizes than the implementation oblivious analysis for the same level of throughput. This confirms our claim that the analysis solely based on SDF operational semantics is overly conservative and yields far larger buffer sizes than required.

In case of `mpeg` application, for example, the implementation oblivious technique reports that a total buffer size of $|\beta| = 15{,}243$ is required to achieve the maximum throughput, while the implementation aware analysis reduces this to $|\beta| = 326$, which is $46\times$ smaller.

Figure 4.8 highlights the substantial reduction in total buffer size requirement, using the same data of Fig. 4.7. The horizontal axis is in logarithmic scale (base 2) and compares the implementation oblivious vs. implementation aware ratio of total buffer size, $|\beta|$, required to achieve the maximum throughput, 80 % of the maximum

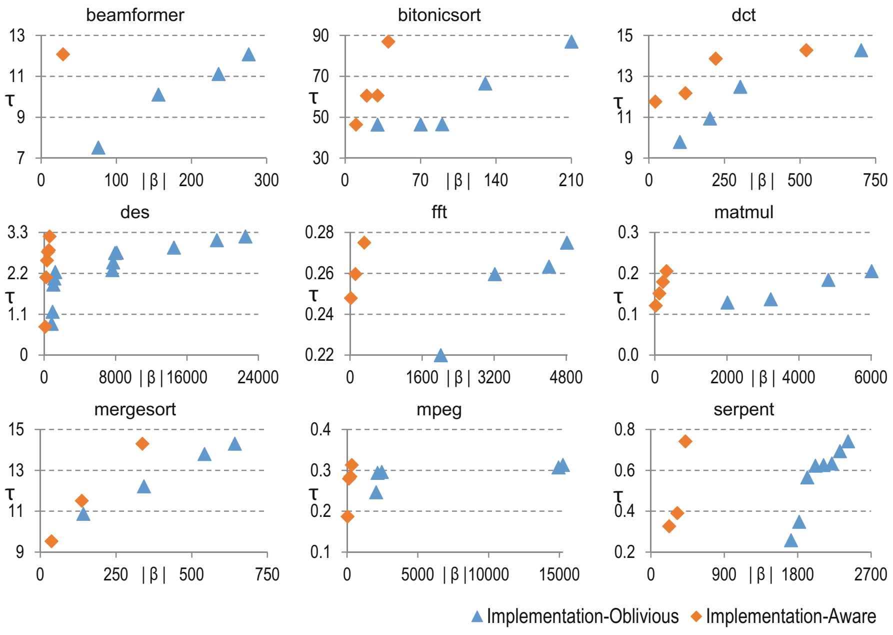


Fig. 4.7 Pareto points between total interprocessor buffer size, $|\beta|$, and the corresponding throughput, τ, for both the baseline implementation oblivious and the proposed implementation aware tradeoff analysis techniques. The proposed method yields substantially improved interprocessor buffer size estimates under identical throughput constraints

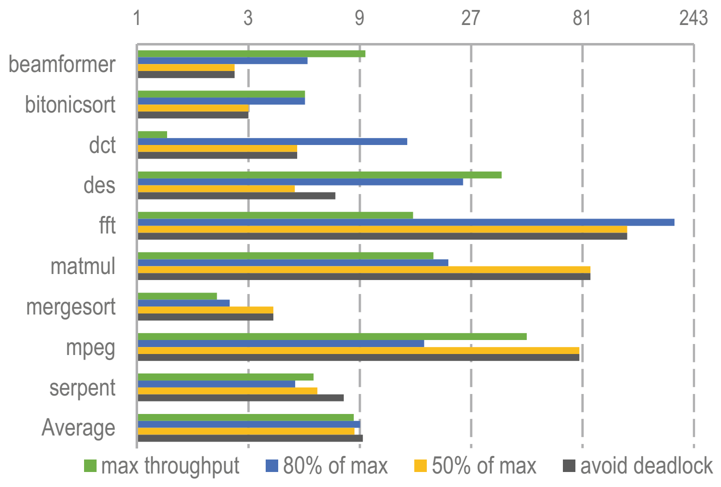


Fig. 4.8 Reduction in total interprocessor buffer size estimates, i.e., the implementation oblivious over implementation aware ratio of total interprocessor buffer size, $|\beta|$, required to achieve the maximum throughput, 80 % of the maximum throughput, 50 % of the maximum throughput, and to avoid deadlock. *X*-axis shows the ratio in base 2 logarithmic scale

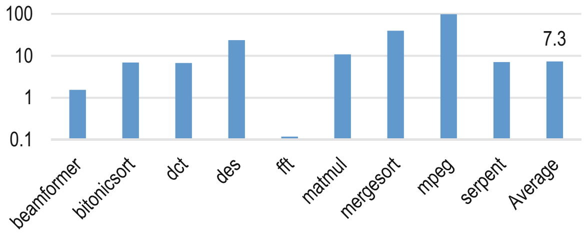


Fig. 4.9 Runtime of implementation aware over implementation oblivious analysis

throughput, 50 % of the maximum throughput, and to avoid deadlock, respectively. On average (geometric mean), using the proposed implementation aware technique, total buffer size $|\beta|$ required to achieve the maximum throughput, 80 % of the maximum throughput, 50 % of the maximum throughput, and to avoid deadlock is reduced by a factor of 8.5, 9.0, 8.5, and 9.3×, respectively.

Figure 4.9 shows how the increase in complexity of the model translates into an increase in the runtime of the analysis. Specifically, it shows the ratio of the time it takes to run the proposed implementation aware tradeoff analysis technique over the time it takes to run the baseline implementation oblivious technique. The ratio heavily depends on the application, e.g., 98× for `mpeg` and 0.11× for `fft` benchmark. On average (geometric mean), the ratio is 7.3×. The workstation employed in our experiments has 8 GB of memory and 3.4 GHz Core i7 processor with 8 MB of cache.

4.2.5.3 Comparison Against Cycle-Accurate Simulation

To quantify the accuracy of estimates produced by the baseline and proposed techniques, we set out to generate executable binaries and simulate their performance under different buffer sizes using the Graphite [18] cycle-accurate simulator (Fig. 4.6c).

Specifically, we utilize StreamIt compiler (RAW processor backend) and generate parallel software code in form of multiple C files from StreamIt SDF applications.[3] We parse the C files and replace generated RAW interprocessor communications with Graphite interprocessor communication API calls. Next, we compile the generated code into binary using `gcc -O2` command, and pass the binaries to Graphite for cycle-accurate simulation (Fig. 4.6c).

For every benchmark, we adjust the buffer size distribution ($\beta(uv)$ for all channels uv) to match buffers that result in the maximum throughput according to

[3]We also experimented with SDF3 benchmarks in Sect. 4.2.5.2. However SDF3 benchmarks merely include graph parameters and not task implementations. Thus, we could only perform the experiments shown in Fig. 4.6a, b and not c. Detailed results are omitted due space limits. For SDF3 benchmarks, on average, buffer size reduction using implementation aware analysis is 6×, and runtime ratio of implementation aware over implementation oblivious is 5×.

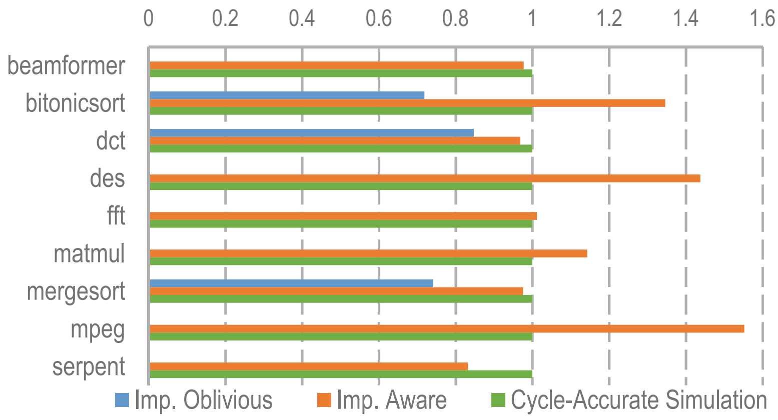


Fig. 4.10 Comparison of (normalized) throughput estimated by implementation aware and implementation oblivious techniques against cycle-accurate simulation. Implementation oblivious technique inaccurately predicts deadlock in most cases, and is less accurate in the remaining cases

implementation aware model analysis. That is, we select buffer size distribution of the orange diamond-shaped point with the highest throughput in every Pareto chart in Fig. 4.7. We have slightly modified Graphite to simulate interprocessor channels with limited buffer size. Since the simulated number of cycles can vary from one application iteration to the next (due to control flow variations, cache effects, etc.), we measure throughput by examining its steady-state long term average. That is, we continue the simulation until no significant change (no more than 1 %) in long term throughput is observed.

Figure 4.10 compares the throughput estimated by implementation aware and implementation oblivious analysis techniques for the selected buffer size distribution, against cycle-accurate simulated throughput. The numbers are normalized with respect to the throughput given by Graphite cycle-accurate simulator. Hence, a value of 1.0 means zero error in estimation of throughput, in comparison with cycle-accurate simulation.

The implementation oblivious analysis falsely reports deadlock ($\tau = 0$) in six out of nine benchmarks. This occurs because the selected buffer sizes are smaller than what implementation oblivious analysis believes to be required for avoiding deadlock. In the other three benchmarks (`bitonicsort`, `dct`, and `mergesort`), the average error is 23 %. The overall average error across all the nine benchmarks using the implementation oblivious analysis technique is 74 %.

The implementation aware analysis, however, estimates the throughput very closely. Compare the orange and green bars in Fig. 4.10. The error in estimation of throughput is less than 5 % in `beamformer`, `dct`, `fft`, and `mergesort` benchmarks. On average, the error of implementation aware analysis in estimation of throughput is 19 %, compared to cycle-accurate simulation.

Let us take a closer look at the `mpeg` benchmark. The implementation oblivious analysis falsely reports deadlock (zero throughput) for this benchmark because the selected buffer sizes are smaller than what implementation oblivious analysis believes to be required for avoiding deadlock. The implementation aware analysis does not perform well either and shows 55 % error in estimation of throughput for the selected buffer sizes. We already know the estimated throughput by both implementation oblivious and implementation aware analysis for different buffer sizes from the Pareto points in Fig. 4.7. We performed cycle-accurate simulations on this benchmark for all buffer sizes in Fig. 4.7 and observed that throughput does not change. Hence both implementation aware and implementation oblivious analysis have 55 % error in estimation of throughput for larger buffer sizes, while the implementation oblivious analysis falsely reports deadlock (zero throughput) for smaller buffer sizes.

Figure 4.11 shows runtime of cycle-accurate simulation over runtime of implementation aware analysis for all benchmarks. The runtime ratio is higher than 100× in six out of nine benchmarks. In the `fft` benchmark the ratio is 606×. On average (geometric mean), it takes about 102× longer to run cycle-accurate simulations than to run the proposed implementation aware analysis.

Comparison with related work is presented in Sect. 4.4. Let us highlight the key benefits offered by the proposed approach. In comparison with implementation oblivious analysis (analysis solely based on SDF operational semantics), it offers substantially more accurate (9× smaller) interprocessor buffer size estimates for the same level of throughput. This is achieved by taking into account very limited information on target implementation. In comparison with cycle-accurate simulation, the implementation aware analysis offers 102× speedup in runtime and relatively low error (19 %) in estimation of throughput. Note that the proposed method is performed at a high-level on SDF graphs, while the cycle-accurate simulation is performed on compiled binary codes and thus, has access to all relevant details, such as processors' instruction set, cache, and program control flow.

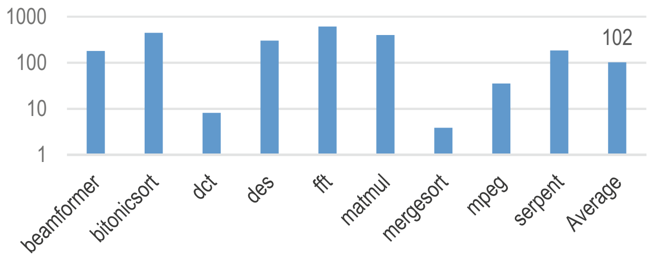


Fig. 4.11 Runtime of cycle-accurate simulation over the proposed implementation aware analysis technique

4.3 Platform-Oriented Throughput Scaling

4.3.1 Overview

In principle, specifying the application as a set of tasks and their dependencies in form of an SDF graph is meant to only model the functional aspects of an application, which should enable seamless portability to new platforms by fresh platform-driven allocation and scheduling of tasks and their executions. However, SDF graphs are rather *rigid* in that some non-behavioral aspects of the application are *implicitly* hard coded into the model at design time. Consequently, allocation and scheduling processes are likely to generate poor implementations when one tries either to port the application to different platforms, or to explore implementation design space on a range of platform choices [38]. The limitations of SDF graphs with portability, scalability, and subsequently the ability to explore implementation tradeoffs (e.g., with respect to number of cores) have become especially critical with availability of platforms with a large number of processor cores, which can dedicate a wide range of resources to an application [5, 47].

As an example, consider merge sort dataflow network, which is composed of actors for splitting the data into segments, sorting of segments using a given algorithm (e.g., quicksort), and merging of the sorted segments into a unified output stream. A specific instance of the sort network would have rigid structural properties, such as number of sort actors or fanin degree of merge actors. The choice of structure, although implicitly hard coded into the specification, is orthogonal to application's end-to-end functionality. It is intuitively clear that the optimal network structure would depend on the target platform, and automatic software synthesis from a rigid specification is bound to generate poor implementations over a range of platforms.

Our driving observation is that the scalability limitation of software synthesis from rigid SDF models could be addressed if the specifications were sufficiently malleable at compile time, while maintaining functional consistency. We present an example manifestation of the idea, dubbed FORMLESS, which extends the classic notion of dataflow by abstracting away some of the unnecessary structural rigidity in the model. In particular, malleable aspects of the dataflow structure are modeled using a set of parameters, referred to as the *forming vector*. Assignment of values to the parameters instantiates a particular structure of the model, while all such assignments lead to the same end-to-end functional behavior. A simple example of a forming set parameter is the fanin degree of merge actors in the sort example.

Our approach opens the door to design space exploration methodologies that can *hammer out* a FORMLESS specification to form an optimized version of the model for the target platform. The "formed" model can be subsequently passed onto conventional allocation and scheduling processes to generate a quality parallel implementation. We also present such a design space exploration scheme that determines the forming set using platform-driven profiles of application tasks. Experimental results demonstrate that FORMLESS yields substantially improved portability and scalability over conventional SDF modeling. Note that the primary objective here is to demonstrate the merit of malleable specifications in terms of scalability, as opposed to development of a formal programming language or a sophisticated design space exploration engine.

4.3.2 Baseline Software Synthesis

Automated parallel software synthesis from SDF models normally involves several key steps that are fairly well researched [4, 11–13, 17, 19, 45]. Figure 4.12 shows the most key steps, namely assignment of tasks to processors, scheduling of tasks for periodic execution on the processors, and code generation. As discussed in later sections, we present our method as an improvement on top of the baseline software synthesis.

Figure 4.13 illustrates an example. Figure 4.13b shows the SDF graph for an example streaming sort application, which sorts 100 data tokens per invocation. The `split` task reads 100 tokens from the input stream, and divides them into two segments of 50 tokens that are passed onto the two `sort` tasks. After the segments are sorted, the `merge` task combines them into the final sorted output

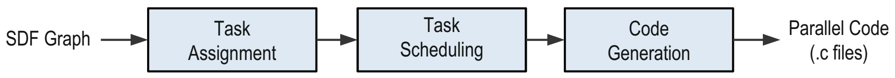
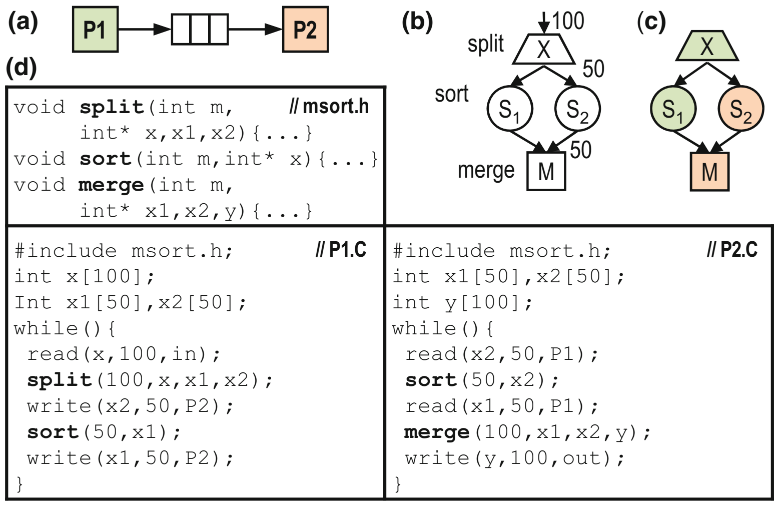


Fig. 4.12 Baseline software synthesis



```
void split(int m,        // msort.h
     int* x,x1,x2){...}
void sort(int m,int* x){...}
void merge(int m,
     int* x1,x2,y){...}
```

```
#include msort.h;        // P1.C
int x[100];
Int x1[50],x2[50];
while(){
 read(x,100,in);
 split(100,x,x1,x2);
 write(x2,50,P2);
 sort(50,x1);
 write(x1,50,P2);
}
```

```
#include msort.h;        // P2.C
int x1[50],x2[50];
int y[100];
while(){
 read(x2,50,P1);
 sort(50,x2);
 read(x1,50,P1);
 merge(100,x1,x2,y);
 write(y,100,out);
}
```

Fig. 4.13 (**a**) Example platform. (**b**) Sort application modeled as SDF graph. (**c**) Tasks are assigned to processors (color coded). (**d**) Synthesized software stream. Figure 4.13c shows an example task assignment, and d the corresponding generated code.

Task functionalities are provided as sequential computations that are kept intact throughout the synthesis process. The software code for each processor is synthesized by stitching together the set of tasks that are assigned to that processor according to their schedule. For tasks that are assigned to the same processor, inter-task communication is implemented using arrays. That is, the producer task writes its data to an array, which is then read by the consumer task. Interprocessor communication is implemented using `read` and `write` system calls.

4.3.3 SDF Limitations in Throughput Scaling

Consider the sort example of Fig. 4.13. We investigate the scaling of throughput when platforms with different number of processors are targeted. Let us assume that the `sort` task implements the quicksort algorithm, and the `merge` task merges two sorted data segments into one stream using the mergesort algorithm.

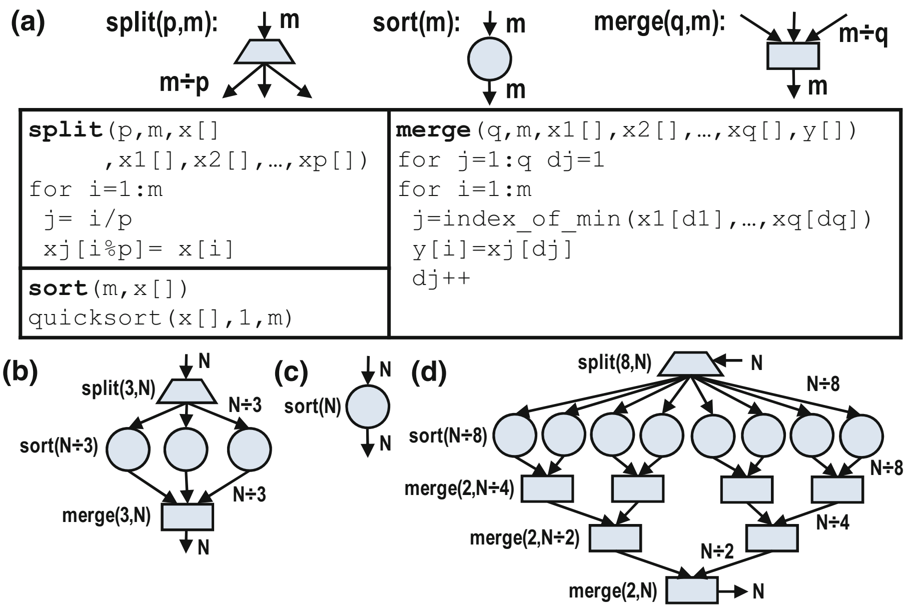
An immediate observation is that the example SDF graph cannot readily utilize many (more than four in the case of depicted SDF graph) processors due to the limited concurrency in the specification. At the other extreme, the throughput of the synthesized software is going to be poor when one processor is targeted, compared to eliminating the split and merge tasks and running a single sort task (i.e., the quicksort algorithm) on the entire input stream.[4] This is partly because the overhead of coordination among multiple parallel tasks is only justified if sufficient amount of parallelism exists in the platform.

Intuitively, increasing concurrency in the SDF graph specification facilitates utilization of more parallel resources and potentially increases the potential for improving performance via load balancing between processors, however, it comes at the cost of degraded performance when platforms with fewer processors are targeted. Conversely, reducing the concurrency in the SDF specification would improve performance on fewer processors at the expense of scalability to platforms with large number of processors.

4.3.4 Proposed Solution: FORMLESS Model

We make the key observation that SDF specifications are structurally rigid and do not fully live up to the intended promise of separating functional aspects of the application from implementation platform, and thus, fail to deliver efficient portability and scalability with respect to number of processors in the platform. To

[4]The discussion does not pertain to sorting of large databases which do not entirely fit in the memory.



```
split(p,m,x[]
     ,x1[],x2[],…,xp[])
for i=1:m
 j= i/p
 xj[i%p]= x[i]
```

```
sort(m,x[])
quicksort(x[],1,m)
```

```
merge(q,m,x1[],x2[],…,xq[],y[])
for j=1:q dj=1
for i=1:m
 j=index_of_min(x1[d1],…,xq[dq])
 y[i]=xj[dj]
 dj++
```

Fig. 4.14 FORMLESS specification of the sort example: (**a**) Actor specifications. (**b**–**d**) Example instantiations

address the portability and scalability limitations, not only application specification has to be sufficiently separated from implementation platform, but it also has to admit platform-driven transformations and optimizations.

Our idea is to specify the tasks and their composition using a number of parameters. Adjustment of parameters enables "massaging" the structure of the SDF graph to fit the target architecture, while all candidate SDF graphs deliver the same end to end functionality. Figure 4.14 sketches the idea for the example sort application in which fanout degree of the split task and fanin degree of the merge task are parametrically specified. The number of tasks, their types, and compositions, as well as their data production rates are immediate functions of the two split-fanout and merge-fanin parameters. Three example instances of the FORMLESS graphs are shown in Fig. 4.14.

4.3.4.1 Formalism

We propose raising the level of abstraction in specifications to eliminate the rigid structure of the SDF graph, while preserving its functional behavior. Our approach is to require application designers to specify the tasks and the structure of the SDF graph using a number of parameters, referred to as the *forming vector*. Specifically, a forming vector Φ is defines as

$$\Phi = (\phi_1, \phi_2, \dots, \phi_{|\Phi|})$$

where ϕ_j is a *forming parameter* whose possible set of values are a subset of domain δ_j. Hence, domain of the forming vector Φ is equal to

$$\Delta = \delta_1 \times \delta_2 \times \cdots \times \delta_{|\Phi|}$$

We extend the definition of a task α such that input ports, output ports, and data transformation function of α are all specified as functions of the underlying parameters in Φ. In other words, task α is defined as the tuple

$$\forall \Phi \in \Delta_\alpha : \ \alpha(\Phi) = \big(\mathrm{In}_\alpha(\Phi), \mathrm{Out}_\alpha(\Phi), F_\alpha(\Phi)\big)$$

For example, the merge task in Fig. 4.14a is defined based on the forming vector $\Phi = \{q, m\}$. The function $\mathrm{In}_{\mathrm{merge}}(q, m)$ specifies q input ports of rate $\frac{m}{q}$, and function $\mathrm{Out}_{\mathrm{merge}}(q, m)$ specifies one output port of rate m. The data transformation function $F_{\mathrm{merge}}(q, m)$ specifies a mergesort algorithm which combines q sorted input arrays of size $\frac{m}{q}$ into a single sorted output array of size m. In this example, $\Delta_{\mathrm{merge}} = \{(q, m) \mid m \geq 2, q \geq 2, m \mod q = 0\}$.

We also extend the definition of SDF graph $G(V, E)$ such that tasks (V) and channels (E) are specified as functions of the underlying parameters in Φ. Formally, SDF graph G is defined as the tuple

$$\forall \Phi \in \Delta_G : \ G(\Phi) = \big(V_G(\Phi), E_G(\Phi)\big)$$

$V_G(\Phi)$ is a function which specifies the set of tasks in G based on forming vector Φ, and is formally defined as

$$V_G(\Phi) = \big\{\alpha_1(\Phi_1), \alpha_2(\Phi_2), \ldots, \alpha_{|V|}(\Phi_{|V|})\big\}$$

where $\alpha_i(\Phi_i)$ is an instance of task α_i which is formed based on forming vector Φ_i, and both α_i and Φ_i are determined based on the given Φ. For instance, the SDF graph in Fig. 4.14b is specified based on forming vector $\Phi = \{p, q, m\} = \{3, 3, N\}$, and function V_G specifies the set of tasks as

$$V_G(3, 3, N) = \big\{\mathrm{split}(3, N), \mathrm{sort}(\tfrac{N}{3}), \mathrm{sort}(\tfrac{N}{3}), \mathrm{sort}(\tfrac{N}{3}), \mathrm{merge}(3, N)\big\}$$

in which, for example, task $\mathrm{merge}(3, N)$ is instance of $\mathrm{merge}(q, m) = \big(\mathrm{In}_{\mathrm{merge}}(q, m), \mathrm{Out}_{\mathrm{merge}}(q, m), F_{\mathrm{merge}}(q, m)\big)$, where $\{q, m\} = \{3, N\}$.

Similarly, $E_G(\Phi)$ is a function which specifies the set of channels in G based on the forming vector Φ, and is formally defined as

$$E_G(\Phi) = \big\{(prd, cns) \mid prd \in \mathrm{Out}_{\alpha_i}(\Phi_i), cns \in \mathrm{In}_{\alpha_j}(\Phi_j)\big\}$$

where (prd, cns) denotes a channel from an output port prd of some task α_i to an input port cns of some task α_j.

We would like to stress that our primary objective in this paper is to demonstrate the merit of the idea and scalability of malleable specifications. In our scheme, it is the programmer's duty to define the ports, task computations, and graph composition based on the parameters. Furthermore, he has to ensure that every assignment of values from the specified domain Δ_G to the forming vector Φ results in the same functional behavior. This tends to be straightforward since tasks perform the same high-level function under different parameters (e.g., splitting, sorting, or merging in the example of Fig. 4.14).

4.3.4.2 Higher-Order Language

Development of a formal higher-order programming language involves many considerations that are beyond the scope of this paper [9, 10, 43]. However, in this section we present an example realization of the general idea that we have developed.

Figure 4.15a presents the prototype for specifying task and application SDF graph based on a set of parameters. The task specification starts with a list of forming parameters and their type. The `interface` section specifies the set of input and output ports of the task, and the `function` section specifies its data transformation function, all based on the given parameters.

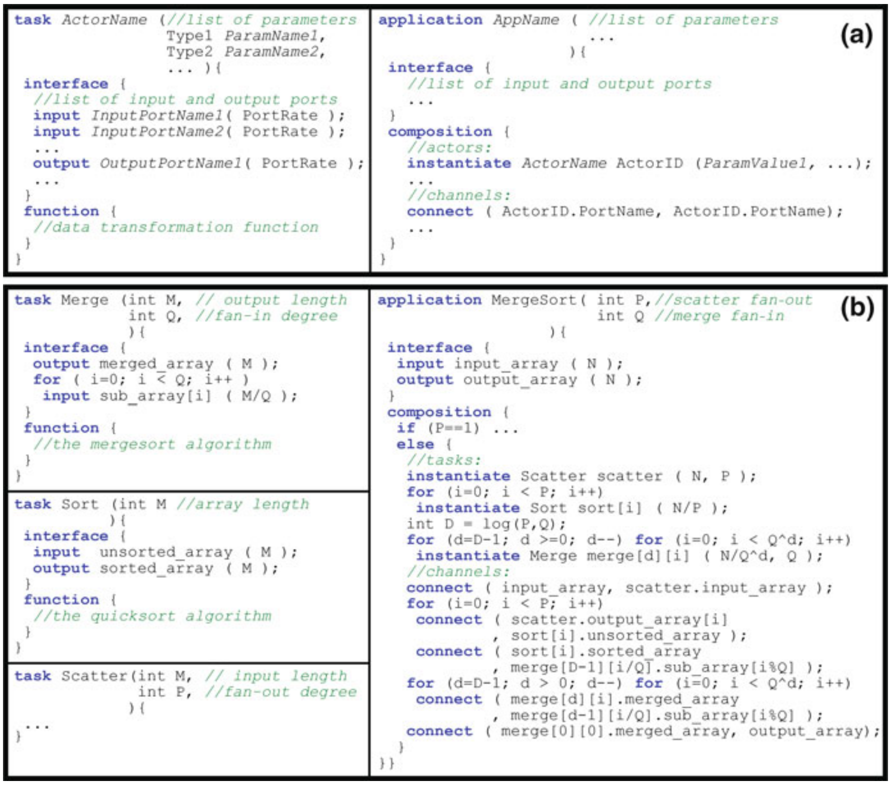
Similarly, application specification also starts with a list of forming parameters. The `interface` section is the same as task interface. In a `composition` section, the tasks are instantiated by assigning the corresponding parameters using the `instantiate` construct. The channels are instantiated using the `connect` construct which connect ports of two tasks.

Figure 4.15b shows the code for our previously mentioned sort application. For example, the merge task is specified with two parameters m and q. As we see the number and rate of input ports in this task is defined using a `for` loop. In general we allow a rich set of programming constructs such as `for` and `if-else` in order to provide enough flexibility in specifying the tasks based on the given forming parameters.

4.3.4.3 Exploration of Forming Parameter Space

To examine the merits of FORMLESS, we developed a design space exploration (DSE) scheme whose block diagram is depicted in Fig. 4.16. The DSE instantiates a platform-driven SDF graph $G(\Phi_{\text{opt}})$ from a given FORMLESS specification by optimizing the forming vector Φ. Central to the quality of the DSE are high-level estimation algorithms for fast assessment of the throughput of a specific instance of the SDF graph.

Task Profiling: The workload associated with a task is composed of two components: computation workload and communication-induced workload. Since tasks are defined parametrically, their computation workload depends on the values of the relevant forming parameters. In addition, computation workload is inherently input-dependent, due to the strong dependency of the tasks' control flow with their



```
task ActorName (//list of parameters
                Type1 ParamName1,
                Type2 ParamName2,
                ... ){
 interface {
  //list of input and output ports
  input InputPortName1( PortRate );
  input InputPortName2( PortRate );
  ...
  output OutputPortName1( PortRate );
  ...
 }
 function {
  //data transformation function
 }
}
```

```
application AppName ( //list of parameters
                      ...
                    ){
 interface {
   //list of input and output ports
   ...
 }
 composition {
   //actors:
   instantiate ActorName ActorID (ParamValue1, ...);
   ...
   //channels:
   connect ( ActorID.PortName, ActorID.PortName);
   ...
 }
}
```

```
task Merge (int M, // output length
            int Q, //fan-in degree
            ){
 interface {
  output merged_array ( M );
  for ( i=0; i < Q; i++ )
   input sub_array[i] ( M/Q );
 }
 function {
  //the mergesort algorithm
 }
}
```

```
task Sort (int M //array length
          ){
 interface {
  input  unsorted_array ( M );
  output sorted_array ( M );
 }
 function {
  //the quicksort algorithm
 }
}
```

```
task Scatter(int M, // input length
             int P, //fan-out degree
            ){
 ...
}
```

```
application MergeSort( int P,//scatter fan-out
                       int Q //merge fan-in
                     ){
 interface {
  input input_array ( N );
  output output_array ( N );
 }
 composition {
  if (P==1) ...
  else {
   //tasks:
   instantiate Scatter scatter ( N, P );
   for (i=0; i < P; i++)
    instantiate Sort sort[i] ( N/P );
   int D = log(P,Q);
   for (d=D-1; d >=0; d--) for (i=0; i < Q^d; i++)
    instantiate Merge merge[d][i] ( N/Q^d, Q );
   //channels:
   connect ( input_array, scatter.input_array );
   for (i=0; i < P; i++)
    connect ( scatter.output_array[i]
            , sort[i].unsorted_array );
    connect ( sort[i].sorted_array
            , merge[D-1][i/Q].sub_array[i%Q] );
   for (d=D-1; d > 0; d--) for (i=0; i < Q^d; i++)
    connect ( merge[d][i].merged_array
            , merge[d-1][i/Q].sub_array[i%Q] );
   connect ( merge[0][0].merged_array, output_array);
  }
}}
```

Fig. 4.15 (**a**) Prototype for specifying task and application. (**b**) An example malleable specification for the sort application in Fig. 4.14

input data. The communication-induced workload exists if some of the producers (consumers) of the data consumed (produced) by the task are assigned to a different processor. We take an empirical approach to characterize the computation workload. We measure the execution latency of several instances of the tasks (based on the forming parameters) on the target processor. For each case, we profile the runtime for several randomly generated input streams to average out the impact of input-dependent execution times. The data is processed via regression testing to obtain latency estimates for all parameter values. Hence, for a task $\alpha(\Phi)$, the profiling data provides DSE with a computation workload W_α.

In addition, for a channel (α, α') with communication volume $N_{(\alpha,\alpha')}$, the communication-induced workload of producer and consumer tasks are analytically characterized as $W_{\text{write}} \times N_{(\alpha,\alpha')}$ and $W_{\text{read}} \times N_{(\alpha,\alpha')}$, respectively. W_{write} and W_{read} are the profiled execution latency of platform communication operations.

SDF Graph Formation: Formation of SDF graph is essentially assignment of valid values to the forming parameters. Any such assignment implies a specific

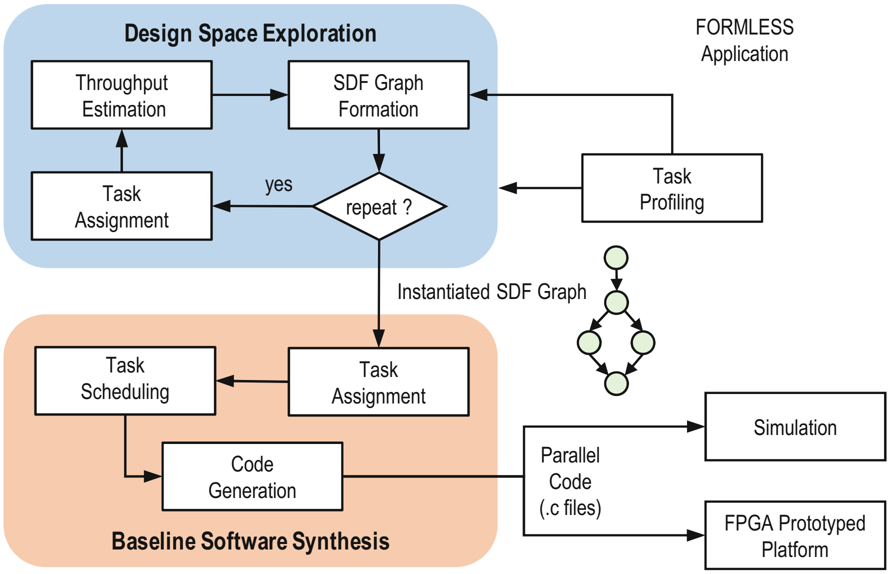


Fig. 4.16 Design space exploration for platform-driven instantiation of a FORMLESS specification

instantiation, which can be passed onto subsequent stages for quality estimation. Our current DSE implementation exhausts the space of forming vector parameters by enumeration, due to the manageable size of the solution space in our testcases, and quickness of subsequent solution quality estimation. In principle, high-level quality estimations can analyze performance bottlenecks to provide feedback and to guide the process of value assignment to forming set parameters. Note that our primary objective in this paper is to demonstrate the scalability of malleable specifications, and not development of a sophisticated DSE.

Task Assignment: Task assignment is a prerequisite to application throughput estimation, and quantifying the suitability of a candidate SDF graph for a target platform. Tasks' computations should be distributed among processors as evenly as possible while interprocessor communication is judiciously minimized. This can be modeled as a graph partitioning problem, in which a graph $G(V, E)$ is cut into a number of subgraphs $G_p(V_p, E_p)$, one for each processor. We employ METIS graph partitioning package [26] for this purpose because our primary focus is to quickly generate solutions to enable integration within the iterative DSE flow. Every vertex (task) $\alpha \in V$ is assigned a weight W_α which denotes its computation workload, and every edge (α, α') is assigned a weight of $N_{(\alpha,\alpha')}$ which denotes its communication volume.

Throughput Estimation: For typical FIFO channels with small latency (relative to processors' execution period), the communication overhead only appears as communication-induced workload on processors. That is, the workload of a processor can be estimated as:

$$W_p = \sum_{\alpha \in V_p} W_\alpha + W_{\text{read}} \times \sum_{\alpha \notin V_p, \alpha' \in V_p} N_{(\alpha,\alpha')} + W_{\text{write}} \times \sum_{\alpha \in V_p, \alpha' \notin V_p} N_{(\alpha,\alpha')}$$

where W_{read} and W_{write} denote the execution latency of platform read and write system calls. The last two terms indicate communication-induced workload on p. We use workload of the slowest processor to estimate the throughput. Formally

$$\text{Throughput} = 1 \div \max_{1 \le p \le P} W_p$$

For a given task assignment, throughput of a candidate solution depends on the buffer sizes of the platform FIFO channels [42], as well as the firing schedule of the tasks that are assigned to the same processor. The above equation merely serves to provide a rough throughput estimate for guiding the DSE. Note that we accurately simulate the impact of interconnect limited buffer size in our final experimental evaluations, which are performed using synthesized software from FORMLESS models in Sect. 4.3.5.

4.3.5 Experimental Evaluation

4.3.5.1 Application Case Studies

To demonstrate the merits of our idea, we experiment with low-density parity check (LDPC), advanced encryption standard (AES), fast Fourier transform (FFT), matrix multiplication (MMUL), and parallel merge sort (SORT).

Low-Density Parity Check: A regular LDPC code is characterized by an $M \times N$ parity check matrix, called the H matrix. N defines the code length and M is the number of parity-check constraints on the code (Fig. 4.17a). Based on matrix H, a Tanner graph is defined which has M check nodes and N variable nodes. Each check node C_i corresponds to row i in H and each variable node V_j corresponds to column j. A check node C_i is connected to V_j if H_{ij} is one (Fig. 4.17b). The input data is fed to the variable nodes, and after processing goes to the check nodes and again back to the variable nodes. This process repeats R times, where R depends on the specific application of the LDPC code. In practice, the H matrix has hundreds or thousands of rows and columns, and therefore, given the complexity of edges in the Tanner graph, we decided not to use this graph as the task graph for software implementation. In fact, direct hardware implementation of the Tanner graph is also not desired because a huge portion of the chip area would be wasted for routing resources [29].

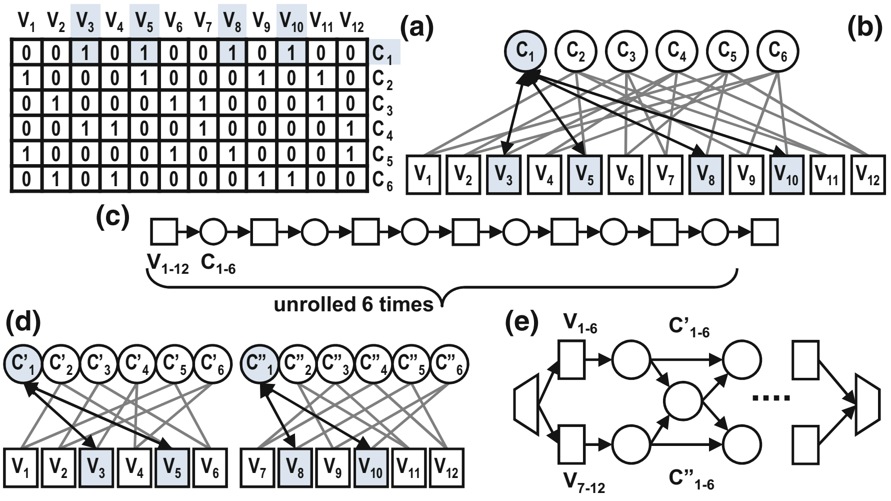


Fig. 4.17 LDPC application: (**a**) Sample H matrix. (**b**) Tanner graph. (**c**) Task graph. Row-Split LDPC based on [29]: (**d**) Tanner graph. (**e**) Task graph

We construct the task graph in the following manner. The variable and check nodes are collapsed into single nodes, and subsequently, the graph is unrolled R times (Fig. 4.17c). We experiment with the LDPC code used in 10GBASE-T standard, where the matrix size is 384×2048 and $R = 6$.

In order to have a malleable specification, we decided to employ the Row-Split method which is a low-complexity message-passing LDPC algorithm and is originally developed for hardware implementation [29]. In this method, in order to reduce the complexity of the edges, the Tanner graph is generated while the rows are split by a factor of $\phi = 2, 4, 8$, or 16. As shown in Fig. 4.17d for $\phi = 2$, the variable nodes are divided into $\phi = 2$ groups, $V_1, \ldots, V_{\frac{N}{2}}$ and $V_{\frac{N}{2}}, \ldots, V_N$, and each check node C_i is split into $\phi = 2$ nodes C'_i and C''_i. The corresponding task graph is shown in Fig. 4.17e, where additional synchronization nodes are required for the check nodes. Interested readers may refer to [29] for further details.

Advanced Encryption Standard: The AES is a symmetric encryption/decryption application which performs a few rounds of transformations on an stream of 128-bit data (4×4 array of bytes). The number of rounds depends on the length of the key which is 10 for 128-bit keys. As shown in Fig. 4.18a, the task graph for the AES cipher consists of four basic tasks called `sub`, `shf`, `mix`, and `ark`. Task `sub` is a nonlinear byte substitution which replaces each byte with another byte according to a precomputed substitution box. In `shf`, every row r in the 4×4 array is cyclically shifted by r bytes to the left. Task `mix` views each column as a polynomial x, and calculates modulo $x^4 + 1$. Task `ark` adds a round key to all bytes in the array using XOR operation. The round keys are precomputed and are different for each of the ten rounds.

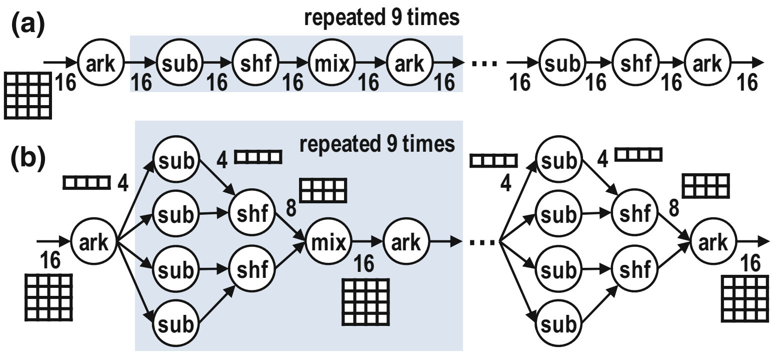


Fig. 4.18 AES: (a) $\Phi = (1, 1, 1, 1)$ (b) $\Phi = (4, 2, 1, 1)$

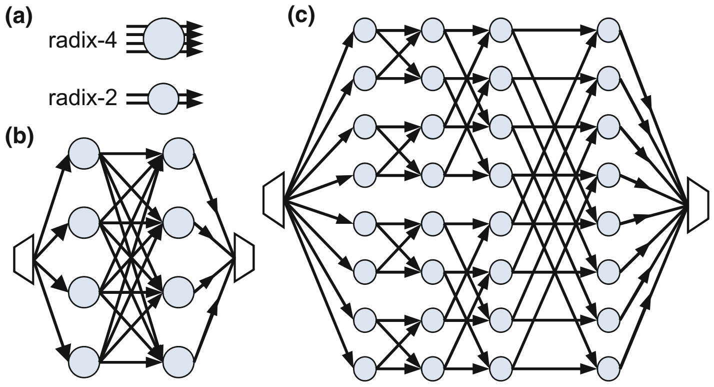


Fig. 4.19 (a) Radix-2 and radix-4 butterfly tasks. (b) 16-point FFT application with radix-4 butterfly tasks. (c) The same FFT computed with radix-2

Therefore, tasks `sub` and `ark` can be parallelized over all elements of the array, and task `shf` only over the four rows, and task `mix` only over the four columns. We constructed the FORMLESS task graph with four parameters. ϕ_1, ϕ_2, and ϕ_4 control the number of rows that the array is divided into for the `sub`, `shf`, and `ark` tasks. Parameter ϕ_3 controls the number of columns that the array is divided into for the `mix` task. For example, the task graph of Fig. 4.18b is formed by $\Phi = (4, 2, 1, 1)$.

Fast Fourier Transform: Fourier transform of an input array is an array of the same size. Fast Fourier Transform (FFT), an efficient algorithm for this computation, is performed using a number of basic butterfly tasks connected in a dataflow network. The basic butterfly operation calculates Fourier of two inputs and is called a radix-2 butterfly. In general, however, FFT can be calculated using butterfly operations with radices other than 2, although typically powers of 2 are used.

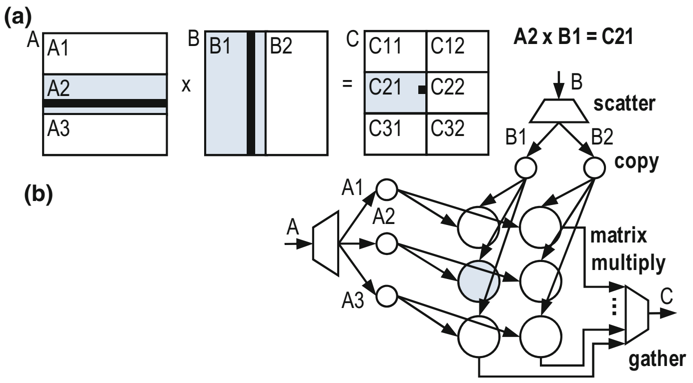


Fig. 4.20 Matrix multiply: (**a**) Block operations for $\Phi = (3, 2)$. (**b**) Task graph formed with $\Phi = (3, 2)$

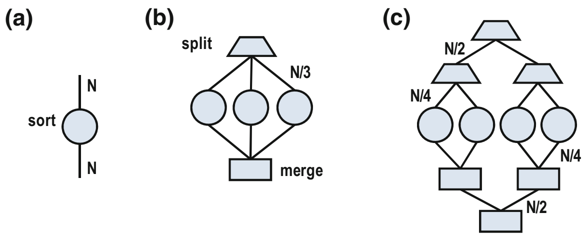


Fig. 4.21 Parallel merge sort: (**a**) $\Phi = (1, 2)$, (**b**) $\Phi = (3, 3)$, (**c**) $\Phi = (4, 2)$

An N-point radix-r FFT uses a dataflow network of radix-r butterfly tasks. This network is organized in $\log_r^N$ stages each containing $\frac{N}{r}$ butterfly tasks. Figure 4.19b shows the structure of the dataflow network for a 16-point FFT application using radix-4 butterfly tasks. Figure 4.19c shows the same computation performed using radix-2 butterflies. Since the computation of FFT is independent of the choice of radix, we define our FORMLESS model for FFT based on a forming parameter ϕ_1 which is the radix. The radix determines structure of the task graph as well as inter-task data communication rates.

Matrix Multiply: The objective is to calculate $A \times B = C$. A block (submatrix) of C can be calculated by multiplying the corresponding blocks of matrix A and B. Adjusting the block size in C trades off the degree of concurrency among operations with the required amount of data replication and movement. Therefore, we construct a FORMLESS task graph with two parameters ϕ_1 and ϕ_2 that control the number of row and column blocks that matrices A and B are divided into. The task graph of Fig. 4.20b is formed by $\Phi = (3, 2)$.

Parallel Merge Sort: A forming parameter ϕ_1 controls the number of parallel sort actors, and a parameter ϕ_2 controls the fanout and fanin degree of the split and merge actors (Fig. 4.21). The value of ϕ_1 should be an integer power of ϕ_2 to generate a valid task graph, i.e., $\phi_1 = \phi_2^n$, $n \geq 0$.

Benchmark	Vector Φ	Domain of Φ
LDPC	(ϕ_1)	$\delta_1 = \{1, 2, 4, 8, 16\}$
AES	$(\phi_1,\ldots,\phi_4)$	$\delta_1 = \ldots = \delta_4 = \{1,2,4\}$
FFT	(ϕ_1)	$\delta_1 = \{2, 4\}$
SORT	(ϕ_1,ϕ_2)	$\delta_1 = \{1, 2, 3, 4, 8, 9, 16, 27\}, \delta_2 = \{2,3\}$
MMUL	(ϕ_1,ϕ_2)	$\delta_1 = \delta_2 = \{1,\ldots, 10\}$

Fig. 4.22 The domain of the forming parameters in our benchmark applications

The domains of the forming vectors used in experimenting the above applications are shown in Fig. 4.22. For example in the AES application, each of the four forming parameters can be 1, 2, or 4.

4.3.5.2 Experiment Setup

We implemented both FORMLESS design space exploration and baseline software synthesis schemes (Fig. 4.16). For a given number of processors, P, within the range of 1 to 100, an optimized task graph $G(\Phi_{\text{opt}})$ is constructed, and subsequently, parallel software modules (separate `.c` files) are synthesized for this task graph.

We consider the following FPGA-prototyped multiprocessor system for throughput measurement of the synthesized software modules. Each processor is an Altera NiosII/f core with 8 kB instruction cache and 64 kB data cache. The communication network is a mesh which connects the neighbor processors with FIFO channels of depth 1024. The processors use a shared DDR2-800 memory, but they only access their own region in this memory, i.e., they communicate only through the FIFO channels. The compiler is `gcc` in Altera NiosII IDE with optimization flag `-O2`.

Due to limited FPGA capacity, we were able to implement the above architecture with up to eight cores. For more number of cores, we employ our previously developed Sequential Execution Abstraction Model (SEAM), which is cycle-accurate in simulating the effect of interprocessor communication (e.g., blocks on empty or full buffers), and also accurately predicts any deadlock situation. We previously confirmed SEAM's accuracy by comparing it with smaller scale multiprocessor systems that we could prototype in FPGA [19, 25].

4.3.5.3 Measurement Results

We compare the throughput of the best instantiated task graph, i.e., $G(\Phi_{\text{opt}})$, with the throughput of rigid task graphs. Figure 4.23 presents the application throughput normalized relative to single-core throughput. The black curves show the throughput values obtained through SEAM simulations from synthesized parallel

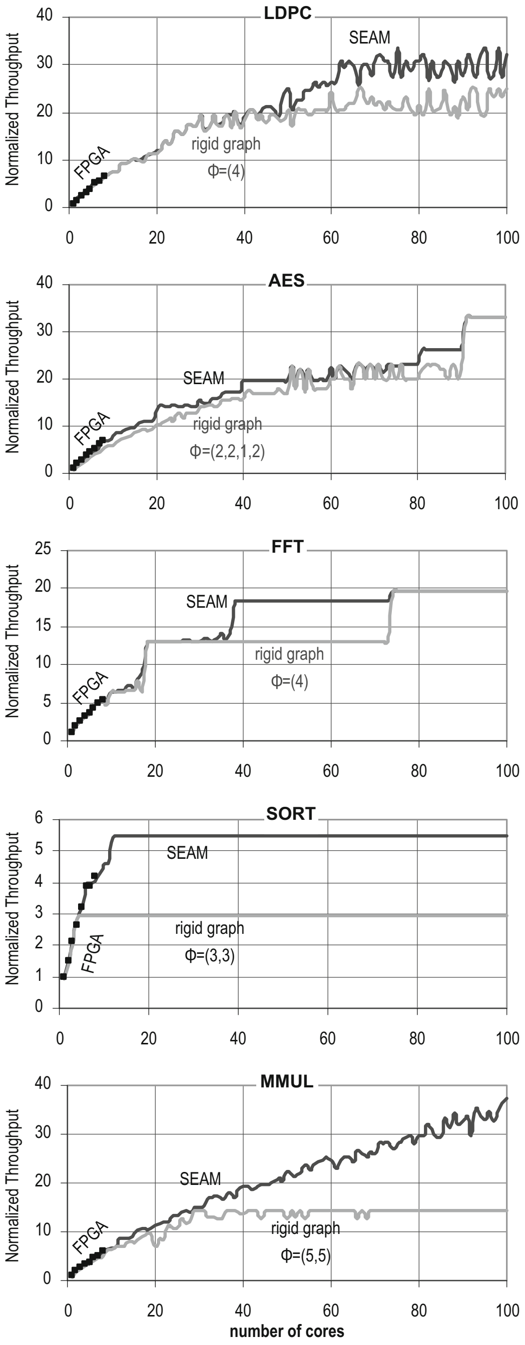


Fig. 4.23 Application throughput on manycore platforms normalized with respect to single-core throughput. The *black curve* shows the throughput obtained from DSE instantiated task graphs. The *gray curves* show the throughput of sample rigid task graphs

implementations, and the 8 black squares show the throughput measured on the FPGA prototype for systems up to eight processors. The gray curves show throughput of a few rigid task graphs selected.

The experiments show that rigid task graphs have a limited scope of efficient portability and scalability with respect to number of processors. For example, an LDPC rigid task graph constructed with forming vector $\Phi = (4)$ does not scale beyond 40 processors. Note that a rigid task graph which scales to large number of processors does not necessarily yields the best throughput in smaller number of processors. For example, an AES rigid task graph constructed with $\Phi = (2, 2, 1, 2)$ only yields the highest throughput for 90 or more processors. For a single processor, this rigid task graph yields 74 % of the best instantiated task graph.

Similar scenarios happen in all benchmark case studies. Each forming vector Φ yields the highest throughput only for a range of targets. In other words, throughput of the best instantiated task graph consistently beats the throughput of any rigid task graph. This result validates the effectiveness of FORMLESS in extending the scope of efficient portability and scalability with respect to number of cores.

It is interesting to see that, for example, in the matrix multiply application $\Phi = (5, 5)$ is not selected for the 25-core target. Instead, the DSE tool selected $\Phi = (6, 4)$ which has 24 multiply tasks. This forming set is not intuitive because one would normally split the multiplication workload into an array of $5 \times 5 = 25$ multiply tasks for 25 cores. The DSE tool considers the effect of smaller tasks (e.g., split tasks), and the communication-induced workloads as well. This again proves that an automated tool outperforms manual task graph formation. However, the DSE is able to scale performance only if the programmer has provided meaningful parallelism. For example, in the SORT application, a larger value for the forming parameter ϕ_1 results in more parallel sort tasks, but the performance does not scale beyond 13 processors because the workload of the terminal merge stage, which is not parallelizable through FORMLESS, becomes the bottleneck.

Figure 4.23 can also be used to determine a reasonable target size, i.e., the number of processors, for each application. For example, in the AES application, more than 40 processors do not yield a throughput gain unless we have at least 50 processors.

4.4 Related Work

Many previous analysis algorithms [2, 16, 31, 40, 41] are solely based on SDF operational semantics in analyzing interprocessor token transfer, i.e., they do not consider the sequential nature of interprocessor token transfer which exists in the underlying implementation.

To increase accuracy in throughput analysis, Moonen et al. [30] proposed to construct a cyclo-static dataflow (CSDF) graph from the given SDF graph by splitting the computation of an SDF task into multiple phases (white box actor model). Our proposed technique in Sect. 4.2, however, focuses on accurate modeling of token production and consumption order (black box actor model), and does not require manual decomposition of task computation.

Oh and Ha [33] proposed a fractional rate model to reduce the buffer size requirement. For applications that work on large blocks of data, e.g., video frames, the dataflow graph is manually transformed into another graph in which actors operate on smaller pieces of data, e.g., one row of a video frame. As a result the buffer requirement is reduced (white box actor model). Our proposed technique in Sect. 4.2 does not require modification of tasks' functional behavior, and treats them as unknown black boxes.

Unfolding an SDF graph [15] is to construct a larger SDF which consists of multiple copies of the original graph. The unfolded SDF has the same functional behavior while expressing more parallelism. This technique can be employed to scale the throughput to manycore systems with larger number of cores. Another technique which also preserves the functional behavior and expresses more parallelism is to convert the input SDF graph to HSDF [15]. The FORMLESS approach introduced in Sect. 4.3 is orthogonal to such techniques and can be applied in parallel with unfolding or conversion to HSDF.

A number of dataflow extensions such as parameterized dataflow [7], scenario-aware dataflow [44], variable-rate dataflow [48], and schedulable parametric dataflow [14] primarily focus on specifications which enable different static and/or dynamic dataflow behaviors based on the parameters. The focus in Sect. 4.3, however, is to specify different possible implementations for the same application behavior, in order to achieve scaling of throughput with respect to the number of processors. We are able to employ a rich set of programming constructs to specify many aspects of the SDF graph based on the forming parameters (Sect. 4.3.4.2). For example, not only the production/consumption port rates but also the number of ports for each task can be specified based on the parameters.

StreamIt compiler [17] automatically detects stateless filters (data-parallel tasks) and judicially parallelizes them in order to achieve better workload balance and hence scaling of performance. This approach provides some level of malleability, but it is limited to data-parallel tasks because it fully relies on the *compiler's* ability to detect malleable sections in the application. In CUDA, scaling of performance is achieved by specifying the application with as much parallelism as practically possible. At runtime, an online scheduler has access to a pool of threads from which the non-blocked threads are selected and executed on available cores [32]. This enables the scaling of performance to newer devices with larger number of processors. In MPI, the programmer may describe the amount of parallelism based on a set of parameters such as the number of available or idle cores. However, since the data rate of communications among MPI processes are not necessarily known at compile time, the allocation and scheduling of the processes are performed by the operating system at runtime.

Our proposed technique in Sect. 4.3 requires the *programmer* to provide a malleable specification, and also, employs *compiler* optimizations to select the best SDF graph based on the malleable specification at compile time.

4.5 Conclusion

We reported two case studies that highlight the potential gap between implementation attributes and those obtained from SDF graph specifications. Specifically, we presented arguments and experiments to demonstrate the weakness of SDF graph-based analysis and synthesis schemes when it comes to optimization of streaming throughput. We discussed the inaccuracy in estimation and the limitation in scaling of streaming throughput, in the context of parallel software synthesis from SDF graphs targeting embedded distributed-memory MPSoCs. Furthermore, we proposed platform-driven extensions to the SDF specification model that alleviate the identified issues. Empirical evaluations confirmed the effectiveness of proposed solutions in improving SDF model fidelity [3, 23].

References

1. M. Ade, R. Lauwereins, J. Peperstraete, Data memory minimisation for synchronous data flow graphs emulated on DSP-FPGA targets, in *Design Automation Conference*, 1997
2. M.A. Bamakhrama, T.P. Stefanov, On the hard-real-time scheduling of embedded streaming applications. Des. Autom. Embed. Syst. Springer Netherlands, **17**(2), 221–249 (2012)
3. K.M. Barijough, M. Hashemi, V. Khibin, S. Ghiasi, Implementation-aware model analysis: the case of buffer-throughput tradeoff in streaming applications, in *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems*, 2015, p. 11
4. S.S. Battacharyya, E.A. Lee, P.K. Murthy, *Software Synthesis from Dataflow Graphs* (Kluwer, Boston, 1996)
5. S. Bell et al., Tile64 - processor: a 64-core soc with mesh interconnect, in *International Solid-State Circuits Conference*, 2008
6. Benchmarks, http://sharif.edu/~matin and http://leps.ece.ucdavis.edu
7. B. Bhattacharya, S. Bhattacharyya, Parameterized dataflow modeling for DSP systems. IEEE Trans. Signal Process. **49**(10), 2408–2421 (2001)
8. S.S. Bhattacharyya, P.K. Murthy, E.A. Lee, *Software Synthesis from Dataflow Graphs* (Springer, Berlin, 1996)
9. J.A. Cataldo, *The power of higher-order composition languages in system design*. Ph.D. thesis, University of California, Berkeley, 2006
10. J.-L. Colaço, A. Girault, G. Hamon, M. Pouzet, Towards a higher-order synchronous data-flow language, in *International Conference on Embedded Software*, 2004, pp. 230–239
11. M.H. Foroozannejad, M. Hashemi, T.L. Hodges, S. Ghiasi, Look into details: the benefits of fine-grain streaming buffer analysis, in *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems*, 2010, pp. 27–36
12. M.H. Foroozannejad, T. Hodges, M. Hashemi, S. Ghiasi, Postscheduling buffer management trade-offs in streaming software synthesis. ACM Trans. Des. Autom. Electron. Syst. **17**(3), 27 (2012)
13. M.H. Foroozannejad, M. Hashemi, A. Mahini, B.M. Baas, S. Ghiasi, Time-scalable mapping for circuit-switched gals chip multiprocessor platforms. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **33**(5), 752–762 (2014)
14. P. Fradet, A. Girault, P. Poplavko, A schedulable parametric data-flow MoC, in *Proceedings of the Conference on Design Automation and Test in Europe*, 2012
15. M. Geilen, Reduction techniques for synchronous dataflow graphs, in *Design Automation Conference*, 2009

16. A.H. Ghamarian et al., Throughput analysis of synchronous data flow graphs, in *International Conference on Application of Concurrency to System Design*, 2006
17. M. Gordon, Compiler techniques for scalable performance of stream programs on multicore architectures. Ph.D. thesis, Massachusetts Institute of Technology, 2010
18. Graphite, http://graphite.csail.mit.edu
19. M. Hashemi, Automated software synthesis for streaming applications on embedded manycore processors. PhD thesis, University of California, Davis, 2011
20. M. Hashemi, S. Ghiasi, Exact and approximate task assignment algorithms for pipelined software synthesis, in *Proceedings of the Conference on Design Automation and Test in Europe*, 2008, pp. 746–751
21. M. Hashemi, S. Ghiasi, Throughput-driven synthesis of embedded software for pipelined execution on multicore architectures. ACM Trans. Embed. Comput. Syst. **8**, 11 (2009)
22. M. Hashemi, S. Ghiasi, Versatile task assignment for heterogeneous soft dual-processor platforms. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **29**(3) (2010)
23. M. Hashemi, M.H. Foroozannejad, S. Ghiasi, C. Etzel, Formless: Scalable utilization of embedded manycores in streaming applications, in *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems*, 2012, pp. 71–78
24. M. Hashemi, M.H. Foroozannejad, S. Ghiasi, Throughput-memory footprint trade-off in synthesis of streaming software on embedded multiprocessors. ACM Trans. Embed. Comput. Syst. **13**(3) (2013)
25. P.-K. Huang, M. Hashemi, S. Ghiasi, System-level performance estimation for application-specific MPSoC interconnect synthesis, in *Proceedings of the 2008 Symposium on Application Specific Processors*, 2008, pp. 95–100
26. G. Karypis, V. Kumar, METIS 4.0: unstructured graph partitioning and sparse matrix ordering system. Technical Report, Department of Computer Science. University of Minnesota, Minneapolis, 1998
27. E.A. Lee, D.G. Messerschmitt, Static scheduling of synchronous data flow programs for digital signal processing. IEEE Trans. Comput. **36**, 24–35 (1987)
28. E.A. Lee, D.G. Messerschmitt, Synchronous data flow. Proc. IEEE **75**(9), 1235–1245 (1987)
29. T. Mohsenin, D. Truong, B. Baas, Multi-split-row threshold decoding implementations for LDPC codes, in *International Symposium on Circuits and Systems*, 2009
30. A. Moonen et al., Practical and accurate throughput analysis with the cyclo static dataflow model, in *International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2007
31. O.M. Moreira, M.J. Bekooij, Self-timed scheduling analysis for real-time applications. EURASIP J. Adv. Signal Process. **2007**, 14 (2007)
32. J. Nickolls et al., Scalable parallel programming with CUDA. ACM Queue **6**, 40–53 (2008)
33. H. Oh, S. Ha, Fractional rate dataflow model for efficient code synthesis. J. VLSI Signal Process. Syst. Signal Image Video Technol. **37**(1), 41–51 (2004)
34. J.D. Owens, U.J. Kapasi, P. Mattson, B. Towles, B. Serebrin, S. Rixner, W.J. Dally, Media processing applications on the imagine stream processor, in *International Conference on Computer Design*, 2002, pp. 295–302 .
35. K. Parhi, *VLSI Digital Signal Processing Systems: Design and Implementation* (Wiley, New York, 2008)
36. A. Pinto, A. Bonivento, A.L. Sangiovanni-Vincentelli, R. Passerone, M. Sgroi, System level design paradigms: Platform-based design and communication synthesis. ACM Trans. Des. Autom. Electron. Syst. **11**(3), 537–563 (2006)
37. A. Sangiovanni-Vincentelli, G. Martin, A vision for embedded systems: platform-based design and software methodology. Des. Test Comput. **18**(6), 23–33 (2001)
38. A. Sangiovanni-Vincentelli, L. Carloni, F. De Bernardinis, M. Sgroi, Benefits and challenges for platform-based design, in *Design Automation Conference, 2004. Proceedings. 41st*, 2004, pp. 409–414
39. SDF3, http://www.es.ele.tue.nl/sdf3

40. S. Stuijk, Predictable mapping of streaming applications on multiprocessors. Ph.D. thesis, Eindhoven University of Technology, The Netherlands, 2007
41. S. Stuijk et al., Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs, in *Design Automation Conference*, 2006
42. S. Stuijk, M. Geilen, T. Basten, Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. IEEE Trans. Comput. **57**(10), (2008)
43. W. Taha, A gentle introduction to multi-stage programming. *Domain-Specific Program Generation* (Springer, Berlin, 2003), pp. 30–50
44. B. Theelen et al., A scenario-aware data flow model for combined long-run average and worst-case performance analysis, in *Proceedings of the International Conference on Formal Methods and Models in CoDesign*, 2006 http://dl.acm.org/citation.cfm?id=2674331
45. W. Thies, Language and compiler support for stream programs. Ph.D. thesis, Massachusetts Institute of Technology, 2009
46. W. Thies et al., Streamit: a language for streaming applications, in *International Conference on Compiler Construction*, 2002
47. D. Truong et al., A 167-processor 65 nm computational platform with per-processor dynamic supply voltage and dynamic clock frequency scaling, in *IEEE Symposium on VLSI Circuits*, 2008
48. M.H. Wiggers, M.J. Bekooij, G.J. Smit, Buffer capacity computation for throughput constrained streaming applications with data-dependent inter-task communication, in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2008
49. Z. Xiao, B. Baas, 1080p h.264/avc baseline residual encoder for a fine-grained many-core system. IEEE Trans. Circuits Syst. Video Technol. **21**, 890–902 (2011)
50. Y. Zhou, E.A. Lee, A causality interface for deadlock analysis in dataflow, in *International Conference on Embedded Software*, 2006, pp. 44–52

Chapter 5
SimSoC: A Fast, Proven Faithful, Full System Virtual Prototyping Framework

Vania Joloboff, Jean-François Monin, and Xiaomu Shi

5.1 Introduction

Computer modeling technologies have become powerful enough that one can build a *virtual prototype* of the system under design. Virtual prototypes make it possible to avoid the tedious and time- consuming process of making real hardware prototypes on which the software has to be tested. They make it possible to develop an emulated system that captures most, if not all, of the required properties of the final system. A virtual prototype of a system under design can be run and tested like the real one, the engineers can exercise and verify the device properties.

In many engineering projects, some components are re-used from former projects, assembled with modified ones, or new ones. It is necessary to support emulation of new hardware components with enough detail, integrated with existing simulation models, possibly coming from third parties. These requirements call for an integrated, modular, full simulation environment where already proven components can be simulated quickly, whereas new components under design can be tested more thoroughly. Modularity and fast prototyping are also important aspects, to investigate alternative designs with easier re-use and integration of third party IP's.

V. Joloboff (✉)
East China Normal University, Shanghai, China

INRIA, France
e-mail: vania.joloboff@inria.fr

J.-F. Monin
Verimag, Université Grenoble Alpes, France
e-mail: jean-francois.monin@imag.fr

X. Shi
Tsinghua University, China
e-mail: xmshi@tsinghua.edu.cn

A. Molnos, C. Fabre (eds.), *Model-Implementation Fidelity in Cyber Physical System Design*, DOI 10.1007/978-3-319-47307-9_5

The `SimSoC` project is developing a framework geared towards full system virtual prototyping, able to simulate complete System-on-Chips or boards. To this end, SimSoC provides a library of simulation models for peripherals, interconnects, together with Instruction Set Simulators (ISS) for the most common embedded systems processors. Each ISS is designed as a SystemC module that issues transactions towards the other models.

This chapter presents the overall system architecture and the `SimSoC` ISS. To achieve fast processor simulation, the ISS technology uses dynamic binary translation. The hardware models are standard SystemC TLM abstractions and the simulator uses the standard SystemC kernel. Therefore, the simulation host can be any commercial off-the-shelf computer and yet provide reasonable simulation performance.

A platform simulator such as `SimSoC` can be used to test critical software such as cryptography or safety related embedded software. In these situations, it is necessary to work with the full system implementation, not only with the model specification. It is also important to use a very faithful simulator in order to ensure that no bias is introduced. To achieve that goal, work was undertaken to demonstrate how it can be verified that the simulated execution of a binary program on the Instruction Set Simulator of a target architecture indeed produces the expected results. This actually requires several steps, to prove first that the translation from C code to machine code is correct, and second that the simulation of the machine code is also correct, that is, they all preserve the semantics of the source code; together with the fact that all of these proofs are verified using a theorem prover or proof checker, not subject to human error in the proof elaboration or verification. The end result is a faithful simulator.

The first part of this chapter describes the generic architecture of the `SimSoC` framework with discussion of speed and accuracy. It illustrates the different simulation modes supported by `SimSoC`, in particular the standard interpretive mode and the two binary dynamic translations modes. It also discusses performance estimation obtained from simulation.

The second part of the chapter describes how a faithful, verified simulator has been developed and proved. The technique presented here partly relies on already existing tools, in particular the Coq proof assistant, the Compcert C compiler, a certified compiler for the C language, combined with our own work to prove the correctness of an ARM Instruction Set Simulator, integrated within SimSoC.

5.2 The SimSoC Framework

In order to simulate a complete hardware platform, one must simulate simultaneously each of the individual components, and possibly advance the clock that represents the simulated execution time. SystemC has become the standard to represent and simulate hardware models, as it is suitable for several levels of abstraction, from functional models to synthesizable descriptions. It is defined by an IEEE

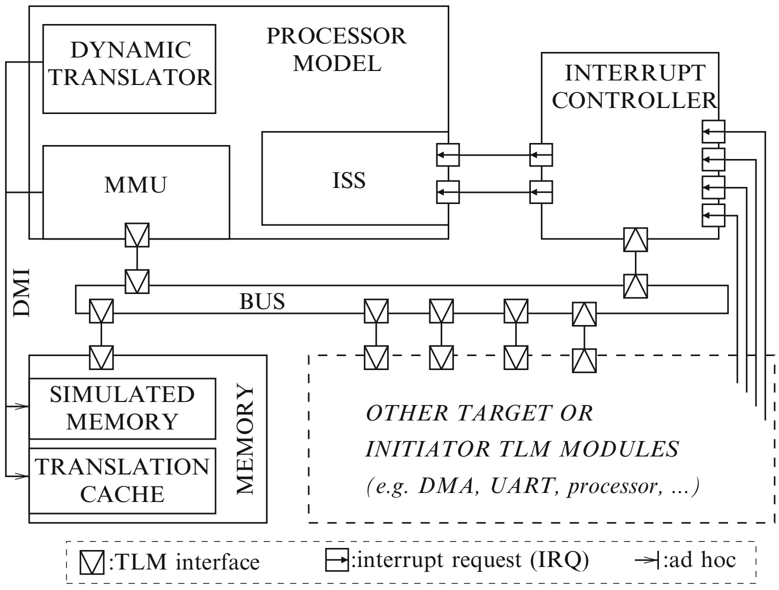


Fig. 5.1 SimSoC architecture

standard [18], and comes with an open-source implementation. Transactional level modeling (TLM) refers both to a level of abstraction [14] and to the SystemC-based library used to implement transactional models [28]. The *transaction* mechanism allows a process of an *initiator* module to call methods exported by a *target* module, thus allowing communication between TLM modules with very little synchronization code.

SimSoC is implemented as a set of SystemC TLM modules and runs on top of the SystemC kernel [16]. The global architecture is depicted in Fig. 5.1. The hardware components are modeled as TLM models, therefore the SimSoC simulation is driven by the SystemC kernel. The interconnection between components is an abstract bus. Each component simulated in the platform is abstracted as a particular SystemC module. Each processor in the target platform is implemented as an ISS that executes its instructions, which may issue transactions towards other components, while maintaining the processor state. A SimSoC ISS simulates the behavior of the processor with instruction accuracy, not cycle accuracy. It emulates the execution of instructions, exceptions, interrupts, and the virtual to physical memory mapping.

The main task of an ISS is to carry out the computations that correspond to each instruction of the simulated program. There are several alternatives to achieve such simulation. In *interpretive simulation*, each instruction of the target program is fetched from memory, decoded, and executed. This technique has been used in popular ISS's such as Simplescalar [6]. It is flexible and easy to implement, but the simulation speed is slow as it wastes a lot of time in decoding. A faster technique to implement an ISS is *dynamic binary cached translation*. With dynamic

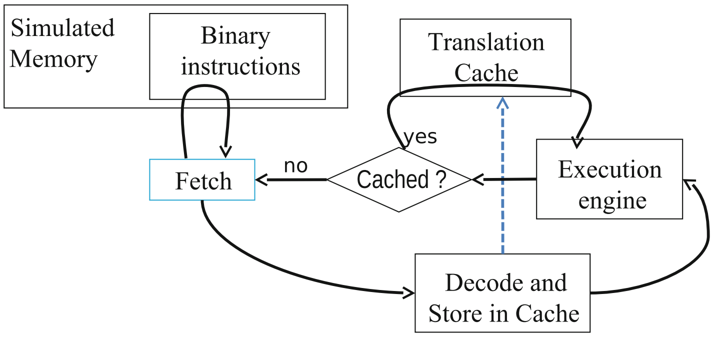


Fig. 5.2 Dynamic binary translation

binary cached translation, illustrated in Fig. 5.2, the target instructions are fetched from memory at run-time but they are decoded only on the first execution and the simulator translates these instructions into another representation, which is stored into a cache. On further execution of the same instructions, the translated cached version is used. If the code is modified during run-time, the simulator must invalidate the cached representation. Although dynamic translation introduces a compile time phase as part of an overall simulation session, this translation latency is amortized over time and it has been commonly used [2, 9, 31, 32, 35]. Dynamic cached translators all work on the same model; however, translation simulators can be subdivided into three categories according to the nature of the translated data and its usage, which we will call here *object oriented*, *code generators*, or *micro-instructions based.*

In a *micro-instructions* based ISS, each instruction is translated into a sequence of pre-defined micro-instructions that are stored into the cache. These pre-defined micro-instructions can be considered as a very low level virtual machine that is run inside the ISS, but this virtual machine is strongly related to the way the simulated processor state is represented inside the ISS. In order to translate each instruction, the parameters from each micro-instruction must be extracted from the instruction (e.g., a constant for an immediate load), and then the micro-instructions must be glued together into a sequence. Because a whole sequence of target instructions can be translated in a block of micro-instructions, the execution is faster and higher simulation speed can be reached. Micro instructions can be compiled in advance with highly optimizing compilers [3]. However, the micro-instructions must be linked together and this process may be dependent upon target binary format and host operating system, hence reducing portability of the ISS.

In the *code generation* technique, the ISS translates the target binary code into native host code. As developing a complete compiler technology is not realistic, the translators tend to leverage off existing compiler technology either (i) by re-generating C code that can be dynamically recompiled for the host machine [30], or (ii) by generating some intermediate language code for which an existing back-end compiler is used to generate the host code [34], for example, generate intermediate representation of the popular GCC compiler and then use GCC back-end to generate the code. The native code generation technique provides higher performance when executing the simulated code. Edinburgh University ISS claims to reach performance in the range of over 400 Mips [20]. However, this performance must be balanced with the throughput. Because dynamic translation is more complex, it takes more time and the simulation time must be long enough to amortize the translation time. This technology is worthwhile when the virtual prototype tests are long enough, and sub-optimal when running very short tests.

In an *object oriented* ISS, the original instructions are translated into a data structure of objects associated with function calls. Each instruction can be translated into one object, each object captures the variable data from the instruction and is bound to a *semantic function* that is called upon execution. With some optimizations a sequence of instructions can possibly be translated into one single object, for example, a basic block. The advantage of an object oriented ISS is that it is not so difficult to construct. In fact using some appropriate input formalism the code of the object classes and methods can often be generated. Using this technique, the ISS is independent of the host operating system and independent of the host processor, therefore easily portable. Because the semantic functions are compiled in advance, a compiler with maximum optimizations can be used and object oriented ISS's can reach speed above 100 Mips on a standard PC computer running at about 3 GHz.

The second mode of `SimSoC` is such an object oriented ISS. In this dynamic translation mode, the decoder dynamically constructs an intermediate representation that maps the binary instructions to a data structure representing the program. The decoding phase mostly amounts to allocating and initializing variables, and locating the appropriate code from the pre-compiled library. In addition, the translator can construct the control flow graph of the decoded software into linked basic blocks, and achieve further optimization at block level.

This mode explores two optimization techniques. First, it offloads the compiling work by pre-compiling most of the simulation code with maximum optimization. Second, it exploits *partial evaluation*, a compiling optimization technique, also known as *specialization* [13]. The basic concept of specialization is to transform a generic program P, when operating on some data d into a faster specialized program Pd that executes specifically for this data. Specialization can be advantageously used in processor simulation, because data can often be computed at decoding time, and a specialized version of the generic instruction can be used to execute it. The simulation code then uses fewer tests, fewer memory accesses, and more immediate instructions. This technique has been used to some extent in the IC-CS simulator [27].

Conceptually, specializing completely the instruction set would mean that for every possible parameter of an instruction (each bit field), there would be a specialized function computing the result. Potentially there are at most 2^{32} specializations of a 32-bit instruction set, which would lead to a huge amount of specialized code. In practice, however, there are reserved encoding bits, many binary configurations are illegal, and overall some instructions are more frequently executed than others.

Fig. 5.3 ARM ADD instruction

```
if ConditionPassed (cond) then
 Rd = Rn + operand
  if S == 1 and Rd == R15 then
    CPSR = SPSR
  else if S == 1 then
    N Flag = Rd[31]
    Z Flag = if Rd == 0 then 1 else 0
    C Flag = CarryFrom(Rn + operand)
    V Flag = OverflowFrom(Rn + operand)
```

By specializing only the most significant parameters and the most frequently used instructions to a higher degree than the less frequent ones, one can reduce the number of specialized functions to a manageable amount of code.

Let us consider the example drawn from the ARM architecture ISS, the `ADD` instruction, noted in assembly language as: `ADD{<cond>}{S} <Rd> ,<Rn>, <operand>`. Its semantic is described in ARM Architecture Reference Manual [33] shown in Fig. 5.3.

The `S` bit indicates whether the instruction updates the Current Process Status Register (CPSR) or not. There are 11 addressing modes used to calculate the operand in an ARM data-processing instruction, such as immediate, using a register, shifted or non shifted, and so on. The operand mode and the addressing mode are specified in the instruction, they are discovered at decoding time. In order to maximize performance, one can use many specialized semantic functions that each carry the very specific task discovered, instead of executing the slower generic one. In order to test multiple specialization possibilities, `SimSoC` includes a generator that can generate the specialized functions automatically based on input specifications. For example, it can specialize the functions on the condition code, the `S` bit, the operand mode, and the registers of data-processing instructions. As an example, the generated semantic function below is the `ADD` instruction simulation code when the condition code is `EQ`, the `S` bit is 0, and the operand is not shifted.

```
PseudoStatus add_Ceq_S0_Mreg
(Processor &proc, PseudoInstruction &pi)
{ if (!proc.cpsr.z) return OK;
  uint32_t tmp = proc.regs[pi.dpi.Rn];
  uint32_t opv = proc.regs[pi.dpi.Rm];
  uint32_t r = tmp + opv;
  proc.regs[pi.dpi.Rd] = r;
  return OK; }
```

In interpretive mode, one function is used to implement the ADD operation. In contrast, in the specialized mode, 330 specialized functions are used to implement ADD corresponding to: 15 (condition code) * 2 (`S` bit) * 11 (operand mode). For the experiment below [17], specialization is used to a limited extent, with no specialization on the registers and no specialization on the condition codes.

Table 5.1 Comparison of interpretive and dynamic translation

	D0	D1	D2
crypto.a0.x	522 s	221 s	57.6 s
arm32, no opt.	6.62 Mips	15.6 Mips	59.9 Mips
crypto.a3.x	139 s	62.2 s	11.6 s
arm32, with opt.	6.84 Mips	15.3 Mips	82.3 Mips
crypto.t0.x	1996 s	576 s	153 s
thumb, no opt.	5.01 Mips	17.3 Mips	65.4 Mips
crypto.t3.x	299 s	90.6 s	26.6 s
thumb, with opt.	5.40 Mips	17.8 Mips	60.7 Mips
loop.a0.x	18.2 s	9.26 s	1.57 s
arm32, no opt.	7.37 Mips	14.5 Mips	85.4 Mips
loop.t0.x	38.1 s	12.1 s	2.49 s
thumb, no opt.	4.84 Mips	15.2 Mips	74.1 Mips

Table 5.1 shows the speed improvement on typical benchmarks between the interpreted mode (D0), a simple dynamic translation cache (D1), and the version using partial interpretation (D2).

To ever increase simulation speed, a third code generation translation mode was added to `SimSoC`, which uses the LLVM [22] library. LLVM is a Low Level Virtual Machine that has been designed to serve as intermediate representation in compilers suitable for complex optimizations. It consists in an abstract instruction set, each instruction having well-defined semantics. An LLVM program can be interpreted directly using the LLVM interpreter, or compiled to machine code. The code generation can be done either with a JIT compiler or a batch compiling phase. LLVM contains a complete set of high-level compiler optimizations, ranging from simple scalar simplifications to complex loop transformations.

The LLVM dynamic translator in `SimSoC` translates on the fly target basic blocks into LLVM functions, yet again using a pre-compiled library in LLVM bitcode. Then one can use the existing LLVM optimizer and Just-In-Time compiler to generate native code, as shown in Fig. 5.4. The translation time from target code to LLVM, next from LLVM to native, can become lengthy and ultimately defeat the speed-up in execution time. Thus, the ISS actually mixes translation modes with a method to evaluate and select only "hot path" code so that the LLVM translation is not systematic, but only operates on such hot paths, the remaining code being simulated with the standard translation. This effectively provides overall faster simulation [19].

In the native code translation mode, the translation-unit is a *Basic block*, a straight sequence of code with only one entry point and only one exit, a branch instruction at the end. By construction, all instructions from a basic block will certainly be executed when it is entered. Each basic block can be compiled into a linear simulation function, which allows fast translation and simulation. Below is an example of a basic block of PowerPC instructions to be translated into an LLVM function:

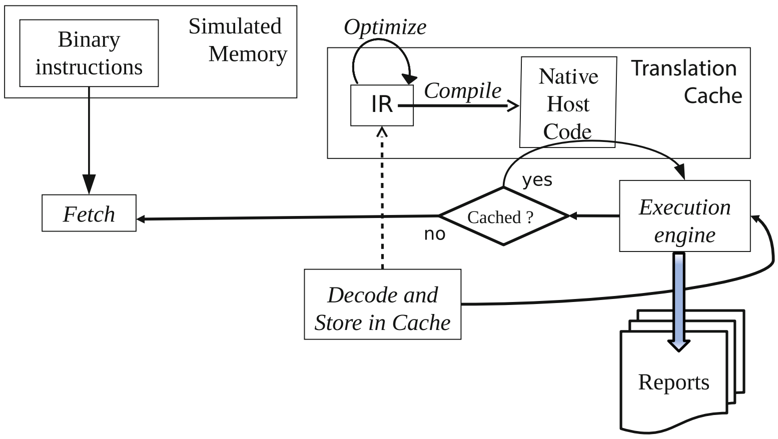


Fig. 5.4 Dynamic binary translation to native code with LLVM

```
addis r9, r0, 385
lwz   r0, 1076 (r9)
or    r1, r0, r0
bl    0xffffff70
```

To translate a basic block to LLVM, the translator creates an LLVM function, containing a single LLVM block `entry`. This LLVM function has a parameter `%proc` that holds the processor state. Then, for each instruction, it generates a call to the corresponding execution function, which must be defined by existing LLVM code. The implementations of these LLVM functions are stored in an LLVM bitcode library, whose generation is explained below. For example, the instructions `addis` and `lwz` are translated to specialized llvm function calls to corresponding functions `addis_ra0` and `lwz_raS`. Each instruction is followed by a function call to update the value of the PC register. The status returned by an execution function tells whether a branch has occurred; by definition, all status but the last tell that no branch occurred. Thus, the basic block above is translated to the following LLVM function.

```
define void @bb_687 (%"struct.Proc"* %proc) {
entry:
 %status = call i32 @addis_ra0(%"struct.
                Proc"* %proc, i8 9, i32 385)
 call void @inc_pc(%"struct.Proc"* %proc)
 %status1 = call i32 @lwz_raS(%"struct.
          Proc"* %proc, i8 0, i8 9, i32 1076)
 call void @inc_pc(%"struct.Proc"* %proc)
 %status2 = call i32 @or(%"struct.
             Proc"* %proc, i8 0, i8 1, i8 0)
 call void @inc_pc(%"struct.Proc"* %proc)
```

```
  %status3 = call i32 @bl(%"struct.
                          Proc"* %proc, i32 -144)
  call void @inc_pc_if_no_branch(i32 %status3,
                          %"struct.Proc"* %proc)
  ret void
}
```

When a basic block has been constructed, one can use LLVM optimization functions at will. In particular, the *AlwaysInline* optimization is systematically called first so that all the code of the execution functions is actually inlined, and thus available for further optimizations. Next, other optimizations can be accomplished. For example, LLVM will reduce the K successive calls to `inc_pc()` inlined functions into a single addition of $K \times 4$ to the PC when the PC variable is never read. In general, after the *AlwaysInline* pass, the LLVM optimization passes named *GVNPass*, *InstructionCombiningPass*, *CFGSimplificationPass*, and *DeadStoreEliminationPass* are also applied.

After the LLVM optimization passes, the LLVM JIT compiler is used to compile LLVM bitcode into host binary code. Then the instruction cache is updated so that the native code is called instead of the simulation function. As it is much easier to write C++ code than LLVM bitcode, to obtain the LLVM library, a library of C++ functions is compiled into an LLVM bitcode library prior to simulation. As an example, here is the C++ code implementing the PowerPC *add* instruction:

```
extern "C" PseudoStatus ppc_add
         (Proc &proc, u8 rt, u8 ra, u8 rb) {
  const uint32_t a = proc.cpu.gpr[ra];
  const uint32_t b = proc.cpu.gpr[rb];
  proc.cpu.gpr[rt] = a+b;
  return OK;
}
```

And here is the LLVM bitcode generated by `llvm-g++`:

```
define i32 @ppc_add(%"struct.Proc"* nocapture
       %proc, i8 zeroext %rt, i8 zeroext %ra,
                  i8 zeroext %rb) nounwind {
entry:
  %0 = zext i8 %ra to i64;
  %1 = geteleptr inbounds %"struct.Proc"*
          %proc, i64 0, i32 2, i32 4, i64 %0;
  %2 = load i32* %1, align 4;
  %3 = zext i8 %rb to i64;
  %4 = geteleptr inbounds %"struct.Proc"*
          %proc, i64 0, i32 2, i32 4, i64 %3;
  %5 = load i32* %4, align 4;
  %6 = add i32 %5, %2;
  %7 = zext i8 %rt to i64;
  %8 = geteleptr inbounds %"struct.Proc"*
           %proc, i64 0, i32 2, i32 4, i64 %7;
  store i32 %6, i32* %8, align 4
  ret i32 0
}
```

Another translation mode with larger translation units has also been experimented, by dynamically determining strongly coupled basic blocks in the control flow graph. Finally, in order to benefit from multi-core simulation hosts, a distributed dynamic translation mechanism has been experimented. In that configuration, the native code translation is achieved by a separate dynamic translation server, that runs concurrently with the ISS on other processors. This work has been described in [37].

5.2.1 Performance Estimate

SimSoC version released in open source is Loosely Timed. It advances the clock using the quantum method. Each instruction is assumed to execute in some time, which defaults to a constant time. Each quantum of N instructions (a parameter) the clock is advanced by the amount of execution time for that quantum. This method makes it possible to run application software with timers and have some idea of the software performance, but it cannot be used for worse case analysis or fine grain performance estimates. However many applications want to have performance estimates.

As cycle accurate simulators are much too slow to be usable in an iterative, agile, development cycle, people have seeked other solution to obtain performance estimates. Attention has turned towards *sampling*, in which a few chunks of code sequences are selected and analyzed. Then statistical methods are used to generate performance estimates. Popular representatives of sampling methods are Simpoint [15], SMARTS [36], and EXPERT [25]. These systems differ mostly in the manner the samples are selected, in size and frequency. Sampling techniques can provide fairly accurate performance prediction with a high probability but may also generate very large data files, and face the issue of initializing state before running the sample.

The idea of "Approximately Timed" is to provide estimates while running entirely the code, not using statistical methods, by exercising abstract models of the architecture, but with a simulation speed that is an order of magnitude faster than a cycle accurate one, yet obtaining measurements that are less than a bounded margin error from the real hardware performance.

Modern processors have complex architectures. They can execute theoretically a certain number of instructions per clock cycle. There are, however, several cases where the instruction flow is disrupted, introducing delays in the computation. Well-known causes of delays are

- There are cache misses. Either data cache miss, or the next instruction to be executed is not available.
- The pipe line is stalled. Instructions are executed in a pipe line to achieve multiple operations in each pipe line issue on each clock cycle, but the pipe line may get stalled in some circumstances.
- There are wait states because of communication with peripherals.

Approximately Timed simulation has been explored in the `SimSoC` project, with the idea to simulate enough of the processes causing the delays to estimate them, though not simulating the exact hardware processes of the caches and pipe line and I/Os. The approach consists in developing a higher abstraction model of the processor (than the CA models) that still execute instructions using fast SystemC/TLM code, but in parallel maintains some architecture state to measure the delays introduced by cache misses and pipe line stalls. These models do not mimic the architecture, they only measure the delays. A case study has been done with a model of the instruction cache and the data cache and the pipe line, for a sample of a Power architecture processor (e200). Our method consists in evaluating the delays with the following approach:

- approximate the delays created by the instruction cache misses, and (pre)fetches. A cache simulator has been implemented that does not simulate the specific hardware architecture cache in detail, but uses an algorithm that reproduces the cache behavior to tell whether there is a cache hit or miss. The instruction buffer is also simulated in an abstract way so that we can compute whether or not the pipe line is fed with instructions.
- build a model of the pipe line architecture that makes it possible to evaluate delays without reproducing in detail the hardware behavior.
- evaluate with a high precision the most frequently executed code (the hot blocks), and use a lower precision for the code that is rarely used (the cold blocks).
- perform a static analysis of basic blocks only once, assuming that future execution of these blocks will approximately take the same number of cycles (except for the cache misses).
- rely on Transaction Level Modeling interface to obtain I/O delays. The delays related to bus arbitration and interconnect access can be captured by TLM transactions as of the TLM 2 standard. A SimSoC ISS relies upon TLM interface to the interconnect to provide timing delays.

This Approximately Timed method purposedly makes errors at least on the following points:

- instruction prefetch is not computed exactly at the same time as the prefetch in the real hardware. In this method the effect of prefetch is computed at block level. For example, it may be the case that at the time of a real prefetch, the interconnect is busy working for other components such as peripherals or coprocessors. In our simulation, the components will use the interconnect but may be at a different time (may be the same, by chance) therefore the delay returned by the interconnect interface is not the real delay that will occur.

- the same is true for memory reads and writes. The cache simulation does not simulate exactly the hardware cache. It only knows when there is cache miss and then generates a transaction. As the interconnect is not solicited at the same time in the simulation than in the real chip, there is a possible error.
- Additional delays introduced by variable length instructions like multiply and divide are ignored. A constant average factor is used.
- As mentioned above, the simulator does not maintain an accurate state of the micro-architecture, there are loopholes in the transition between basic blocks.

This has resulted in good approximation while reducing the simulation speed in acceptable manner for the Power e200 processor model used in our case study. For the software developers who want to quickly compare various software/hardware implementations, fixing all of these error sources would take a very significant simulation overhead, whereas the error introduced fits in the approximation objectives.

5.3 Faithful Simulation

5.3.1 Objective

In many applications nowadays, virtual prototyping is used to design, develop, and test new applications. Most of these virtual prototypes include an Instruction Set Simulator (ISS) to simulate the target processor. The ISS runs the target executable binary code in emulating the hardware and generates the outputs that the executable should produce when run on the target platform. An ISS can be used, for example, to optimize algorithms such as cryptographic software, or to debug new compiler developments, or in the design of many embedded systems applications. Instead of using real hardware prototypes, simulated platforms are more convenient and less expensive. Then, it is important to be sure that the simulator used is faithful to the hardware that it emulates. A *faithful ISS* must produce exactly the same results as the executable would if run on hardware implementation of the instruction set, and this guarantee must be proven.

We have started a first attempt to formally verify that the execution of a program on our Instruction Set Simulator for the target ARM architecture indeed produces the expected results, to be certain that the data output from the simulator, the final processor and memory states are indeed identical to the result obtained with the real hardware. This requires sequential steps, to prove first that the translation from the C code of the simulator to the simulation machine is correct, and second that the simulation of the target machine code is also correct, that is, it preserves the semantics of the computer architecture, together with the fact that all of these proofs are verified using a theorem prover, or proof checker, not subject to human error in the proof elaboration or verification.

The sections below are organized as follows. Section 5.3.2 provides the formal verification background in our context. Section 5.3.3 describes the tools that have been used, in particular the Compcert C compiler, a certified compiler for the C language, and the Coq proof assistant. The proof structure presented afterwards sketches our contribution to prove the correctness of an ARM Instruction Set Simulator. In summary, the method consists in proving each instruction of the instruction set independently, by proving that the execution of the C code simulating an instruction yields identical result to that obtained by a formal executable model of the architecture.

Each independent proof requires using a number of lemmas from a generic lemmas library and usage of a new inversion tactics in the theorem prover. Finally, our conclusion mentions lessons learned and directions for future work.

5.3.2 Formal Verification Background

Program certification has to be based on a formal model of the program under study. Such a formal model is itself derived from a formal semantics of the programming language. Axiomatic semantics and Hoare logic have been widely used for proving the correctness of programs. For imperative programming languages such as C, a possible approach is to consider tools based on axiomatic semantics, like `Frama-C` [8], a framework for a set of interoperable program analyzers for C. Most of the modules integrated inside rely on ACSL (ANSI/ISO C Specification Language), a specification language based on an axiomatic semantics for C.

`Frama-C` software leverages off from `Why` technology [5, 11], a platform for deductive program verification, which is an implementation of Dijkstra's calculus of weakest preconditions. `Why` compiles annotated C code into an intermediate language. The result is given as input to the VC (Verification Conditions) generator, which produces formulas to be sent to both automatic provers or interactive provers like Coq.

In our case of verifying an instruction set implementation, one has to deal with a very large specification including complex features of the C language. A framework is required that is rich enough to make the specification manageable, using abstraction mechanisms for instance, and in which an accurate definition of C features is available. To verify specific properties referring to a formal definition of the ARM architecture, operational semantics offer a more concrete approach to program semantics as it is based on states. The behavior of a piece of program corresponds to a transition between abstract states. This transition relation makes it possible to define the execution of programs by a mathematical computation *relation*. This approach is quite convenient for proving the correctness of compilers, using operational semantics for the source and target languages (and, possibly intermediate languages).

Operational semantics are used in `CompCert` (described below) to define the execution of C programs, or more precisely programs in the subset of C considered by the `CompCert` project. The work presented in this paper is based on this approach. Interesting examples are given by Brian Campbell in the CerCo project [7], in order to show that the evaluation order constraints in C are lax and not uniform.

A very significant verification work has been done to prove the SEL4 operating system [21]. It is comparable to our work in that they have considered a C implementation. The main difference is that they have not considered operational semantics of C, but deduced the proof obligations from the C code, considering the compiler and the architecture as correct. In our work, we believe that the subset of C accepted by `CompCert` is even larger than the subset accepted in SEL4.

Regarding formalization and proofs related to an instruction set, a Java byte code verifier has been proved by Cornelia Pusch [29], the Power architecture semantics has been formally specified in [1], and closer to our work, the computer science laboratory in Cambridge University has used HOL4 to formalize the instruction set architecture of ARM [12]. The objective of their work was to verify an implementation of the ARM architecture with *logical gates*, whereas we consider an ARM architecture simulator coded in C. Reusing the work done at Cambridge in [12] was considered. But, because our approach requires a certified C compiler, in this case `CompCert` C, which is itself coded in Coq, it would have required us to translate all of the C operational semantics as well, which would have been error prone, not to mention the very large effort. It was more convenient to develop our formal model and our proofs in Coq.

5.3.3 Background Tools

The formal verification of the ISS is achieved by using two other existing software tools that have been themselves verified, namely Coq and CompCert C.

5.3.3.1 Coq

Coq [4] is an interactive theorem prover, implemented in OCaml. It allows the expression of mathematical assertions, mechanically checks proofs of these assertions, helps to discover formal proofs, and may extract a certified program from the constructive proof of its formal specification. Coq can also be presented as a dependently typed λ-calculus (or functional language). For a detailed presentation, the reader can consult [10] or [4]. Coq proofs are typed functions and checking the correctness of a proof boils down to type-checking.

The logic supported by Coq includes arithmetic, therefore it is too rich to be decidable. As full automation is not possible for generating proofs, human interaction is essential. The latter is realized by *proof scripts*, which are sequences of commands for building a proof step by step. Coq also provides built-in *tactics* implementing various decision procedures for suitable fragments of the calculus of inductive constructions and a language which can be used for automating the search of proofs and shortening scripts.

When a proof has been interactively developed, Coq automatically verifies the proof, or possibly signals where errors are located. Our work has consisted in developing proofs demonstrating that the C functions simulating the behavior of the ARM processor indeed implement the ARM architecture semantics.

5.3.3.2 Compert-C

`CompCert` is a formally verified compiler for the C programming language provided by INRIA [23, 24], which currently targets Power, ARM, and 32-bit x86 architectures. The compiler is specified, programmed, and proved in Coq. It aims to be used for programming embedded systems requiring high reliability. The generated assembly code is proved to behave exactly the same as the input C program, according to a formally defined operational semantics of the language.

A key point is that we are considering here C programs compliant with the definition of ISO-C 99 standard of *correct C programs*. Indeed the ISO-C standard identifies many constructions that are syntactically correct, but have undefined semantics such as `a[i++] = i;`. The document identifies about one hundred such constructions, and says that a C compiler in that case basically may choose its own interpretation of the abstract syntax, resulting in *unspecified behavior*. This is very important in our work. All of the C code implementing the ISS is correct with respect to the ISO C standard, meaning that it does not contain any construction with unspecified behavior. Compcert-C does not accept such ill-defined expressions and only well-formed programs can be translated according to the formal, unambiguous, semantics. All of the C code considered here has unique and well-defined semantics. We need to prove that it implements the ARM semantics, but we do not need to worry about multiple interpretations.

Three parts of `CompCert` C are used in this work. The first is that we use the correct machine code generated by the C compiler. The second is the C language operational semantics in Coq from which we get a formal model of the program. Third, we use the `CompCert` Coq library for words, half-words, bytes, etc., and bitwise operations to describe the instruction set model. These low level functions have been proven already in `CompCert`, so we can safely re-use them.

It must be noted that the C code of an ISS does not use functions from the C library that invoke the operating system, such as `gettimeofday()`. It uses a very limited number of functions from the C library such as `memset()` or `memcpy()`. `CompCert` provides the formalized properties of such built-in external functions, so we can reason formally on their potential side effects in our proofs.

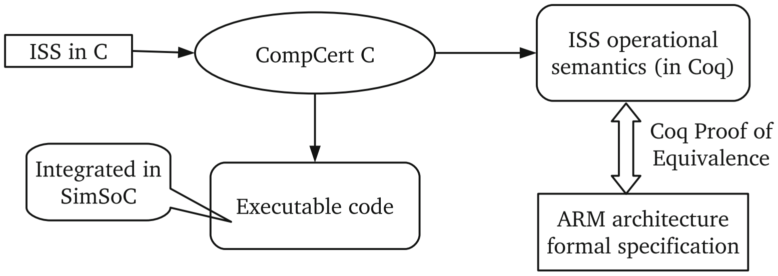


Fig. 5.5 Overall goal

5.3.4 Verified Simulation

The general objective is to obtain a verified simulator that is illustrated in Fig. 5.5. Considering the ARM architecture, we need to have the following:

- a formal model of the ARM instruction set.
- an instruction set simulator of the ARM architecture coded in the (`CompCert`) C programming language.
- a formal operational semantics of the ISS. As shown in Fig. 5.5, from the ISS source code in C, we can obtain through `CompCert` C on one hand the Coq formal semantics of the compiled C program constructed by `CompCert`, since the intermediate representation of the C compiler is a Coq representation and, on the other hand, the verified machine code, which conforms to this operational semantics as guaranteed by CompCert. We use both, the compiled machine code to run simulations and the formal semantics for the proof.
- prove, using the Coq proof assistant, that the resulting ISS semantics indeed implement the formal model of the ARM processor, which boils down to verifying that the semantics of the simulator accurately modifies the processor (and memory) state representation at each step and ends up in results that comply with the formal model of the ARM architecture.

  These steps are described in the following paragraphs.

5.3.4.1 Constructing the Formal Model

Ideally the formal specification of the ARM architecture should be provided by the vendor. But it is not the case, an issue already raised in the work with HOL4 mentioned above [12]. We decided to derive the formal model of ARM architecture in Coq from the architecture reference manual as output of a semi-automated process. The main relevant chapters of the manual are

- `Programmer's Model` introduces the main features in ARMv6 architecture, the data types, registers, exceptions, etc;

- `The ARM Instruction Set` explains the instruction encoding in general and puts the instructions in categories;
- `ARM Instructions` lists all the ARM instructions in ARMv6 architecture in alphabetical order and the `ARM Addressing modes` section explains the five kinds of addressing modes.

There are 147 ARM instructions in the ARM V6 architecture. For each instruction, the manual provides its encoding table, its syntax, a piece of pseudo-code explaining its own operation, its exceptions, usage, and notes. Three kinds of information are extracted for each ARM operation: its binary encoding format, the corresponding assembly syntax, and the instruction semantics, which is an algorithm operating on the processor state. This algorithm may call basic functions defined elsewhere in the manual, for which we provide a Coq library defining their semantics. Other than these extracted data files, there is still useful information left in the document which cannot be automatically extracted, such as validity constraints information required by the decoder generator. However, the most tedious (then, arguably, error prone) part is described using fairly simple, precise and regular pseudo-code, allowing us to extract the Coq formal model in three automated steps: (i) extracting information from the `.pdf` file; (ii) parsing the data into abstract syntax trees, and (iii) automated translation from the abstract syntax into Coq formal model.

During this process, a dozen documentation problems were found but none that were relevant to instruction semantics. These documentation mistakes have been acknowledged by ARM Ltd. Moreover, a single mistake in our automated extractor would impact the formal model of many or even all instructions and then become rather easy to detect. The model has then tested on real programs to verify that we obtain the same results, which gives reasonable confidence in the model.

5.3.4.2 Proof Structure

The proof starts from an ISS coded in C, where each instruction is coded as a C function that modifies the processor state and possibly the memory state (but everything is represented in memory on the simulation host machine). Each C function may also call basic functions from a library. As mentioned above, this C code does not include any construction with "unspecified behavior" of the C language specification. To prove that the simulator is correct, we need to prove that, given the initial state of the system, the execution of an instruction as implemented by a C function results in the same state as the formal specification. To establish the proof, a formal model of that C implementation is provided by `CompCert`, which defines operational semantics of C formalized in Coq.

The proof shall demonstrate that the operational semantics of the C code corresponds to the ARM formal specification. The complete proof is too lengthy for this article, and we only provide here an outline of the method. The state of the ARM V6 processor defined in the formal model is called the *abstract state*.

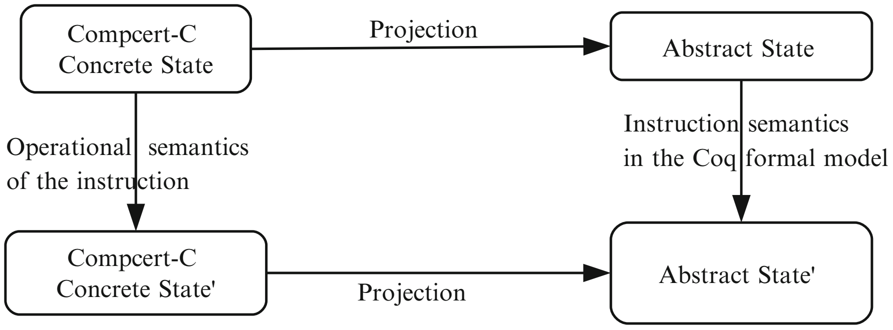

Compcert-C Concrete State
Projection
Abstract State
Operational semantics of the instruction
Instruction semantics in the Coq formal model
Compcert-C Concrete State'
Projection
Abstract State'

Fig. 5.6 Theorem statement for a given ARM instruction

Alternatively, the same state is represented by the data structures corresponding to C semantics that we shall call the *concrete state*. In order to establish correctness theorems we need to relate these two models. Executing the same instruction on the two sides produces a pair of new processor states which should be related by the same correspondence. Informally, executing the same instruction on a pair of equivalent states should produce a new pair of equivalent states, as schematized by Fig. 5.6. Equivalent states are defined according to a suitable projection from the C concrete state to the abstract model, as represented in Fig. 5.7.

5.3.4.3 Projection

In order to achieve a high speed simulation, the C ISS includes optimizations. In particular, processor state representation in the C implementation is complex, not only due to the inherent complexity of the C language memory model, but also because of optimization and design decisions targeting efficiency. Despite the complexity of the C memory model, the `CompCert` C semantics makes it possible to define and prove the projection function. Fortunately, all of the instructions operate on the processor state and there is a single representation of that state in the simulator. It is necessary and sufficient to prove the projection for each particular case of the representation structure. For example, the projection of a register performs a case analysis on possible values, whereas the projection of saved data upon exceptions depends on the type of exception modes. Although there are a number of specific cases to handle, the proof of the projection is relatively straightforward. In more detail:

- The C implementation uses large embedded *struct*s to express the ARM processor state. Consequently the model of the state is a complex Coq record type, including not only data fields but also proofs to verify access permission, next block pointer, etc.

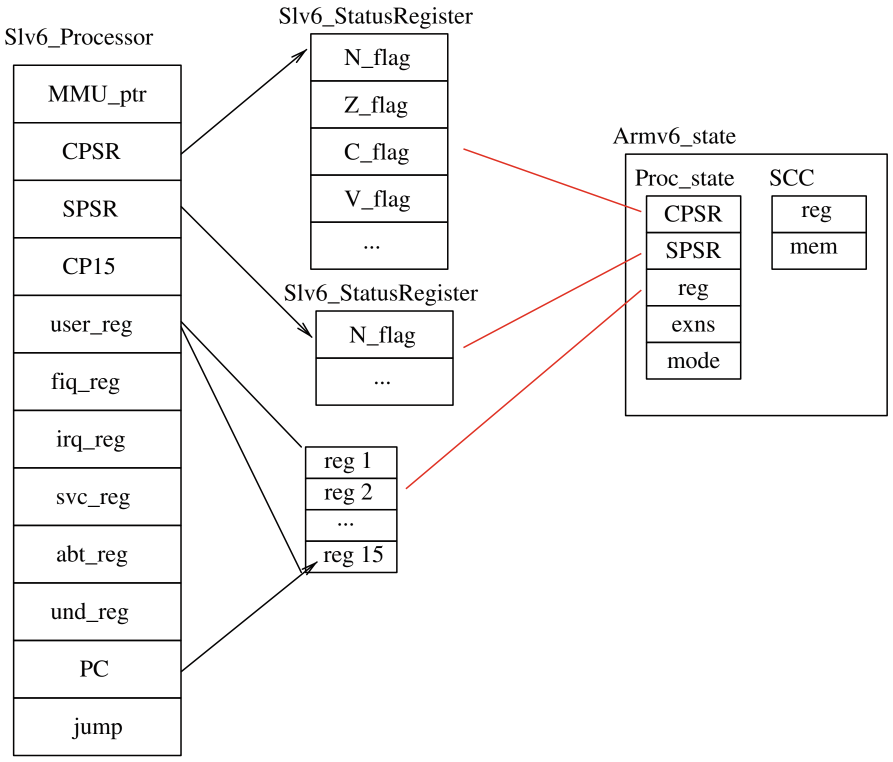


Fig. 5.7 Projection

- Transitions are defined with a relational style (as opposed to a functional style where reasoning steps can be replaced by computations). Relational style is more flexible, especially when dealing with constraints; and fits well with operational semantics.
- The global state is based on a memory model with load and store functions that are used for read/write operations.

The proofs for instructions start from the abstract state described by the formal specification. To verify the projection of the original state, we need the following data: the initial memory state, the local environment, and the formal initial processor state. The projection is meaningful only after the C memory state is prepared for evaluating the current function body representing an ARM instruction. In the abstract Coq model, we directly use the processor state `st`. But on the C side, the memory state is described by the contents of several parameters, including the memory representation of the processor state. We also need to observe the modifications of certain memory blocks corresponding to local variables.

The semantics of `CompCert` C considers two environments. The global environment *genv* maps global function identifiers, global variables identifiers to their blocks in memory, and function pointers to a function definition body. The local environment *env* maps local variables of a function to their memory blocks reference. It maps each variable identifier to its location and its type, and its value is stored in the associated memory block. The value associated with a C variable or a parameter of a C function is obtained by applying `load` to the suitable reference block in memory. These two operations are performed when a function is called, building a local environment and an initialized memory state. When the program starts its execution, *genv* is built. The local environment *env* is built when the associated function starts to allocate its variables. Therefore, on the concrete side, a memory state and a local environment are prepared initially using two steps. First, from an empty local environment, all function parameters and local variables are allocated, resulting into some memory state and the local environment. Second, function parameters are set up using a dedicated function `bind_parameters` and the initial state is thus created.

5.3.4.4 Lemmas Library

Next, we need to consider the execution of the instruction. In the C ISS, there is a standalone C function for each ARM V6 instruction. Each function (instruction) has its own correctness proof. Each function is composed of its return type, arguments variables, local variables, and the function body. The function body is a sequence of statements including assignments and expressions. Let us consider as an example the ARM instruction `BL` (`Branch and Link`). The C code is

```
void B(struct SLv6_Processor *proc,
       const bool L,
       const SLv6_Condition cond,
       const uint32_t signed_immed_24){
 if (ConditionPassed(&proc->cpsr, cond)){
  if ((L == 1))
   set_reg(proc,14,address_of_next_instruction(proc));
   set_pc_raw(proc,reg(proc,15)+(SignExtend_30
                          (signed_immed_24)<<2));
 }
}
```

`CompCert` has designed semantics for `CompCert` C in big-step inductive types for evaluating expressions, which we re-use for the proof. The semantics is defined as a relation between an initial expression and an output expression after evaluation. Then, the body of the function is executed. On the concrete side, the execution yields a new state **mfin**. On the abstract side, the new state is obtained by running the formal model. We have to verify that the projection from the concrete state **mfin** is related to this abstract state. The proof is performed in a top-down manner. It follows the definition of the instruction, analyzing the expression step by step. The function body is split into statements and then into expressions. When evaluating an expression, one has to search for two kinds of information. The first one is how the memory state changes on the concrete side; the other is whether the results on the abstract and the concrete model are related by the projection. To this end, a library of lemmas had to be developed, identifying five categories summarized below.

1. *Evaluating a* `CompCert` *expression with no modification on the memory state.* Such lemmas are concerned with the expression evaluation on `CompCert` C side and in particular the C memory state change issue. Asserting that a memory state is not modified has two aspects: one is that the memory contents are not modified; the other is that the memory access permission is not changed. For example, evaluating the boolean expression $Sbit\ == \ 1$ returns an unchanged memory state.

$$\begin{array}{l}\text{if } G,E \vdash \texttt{eval\_binop}_c\ (Sbit\ ==\ 1),\ M \overset{\varepsilon}{\Longrightarrow} v,\ M' \\ \text{then } M = M'.\end{array}$$

   In Coq syntax, the relation in premise is expressed with `eval_binop`. In this lemma and the following, E is the local environment, G is the global environment, M is the memory state, ε is the empty event (we may have here a series of events, e.g., system call, volatile load/store), and v is the result. The evaluation is performed under environments G and E. Before evaluation, we are in memory state M. With no event occurring, we get the next memory state M'. According to the definition of `eval_binop`, an internal memory state will be introduced.

$$\frac{G,E\ \vdash a_1, M \Rightarrow M' \quad G,E\ \vdash a\_2, M' \Rightarrow M''}{G,E\ \vdash (a_1\ binop\ a\_2), M \Rightarrow\ M''}$$

   In the example, expression a_1 is the value of $Sbit$ and a_2 is the constant value 1. By inverting the hypothesis of type `eval_binop`, we obtain several new hypotheses, including on the evaluation of the two subexpressions and the introduction of an intermediate memory state M''. Evaluating them has no change on the C memory state, hence we have $M = M'' = M'$. In more detail, from the `CompCert` C semantics definition, we know that the evaluation of an expression will change the memory state if the evaluation contains uses of `store_value_of_type`. In `CompCert`, the basic store function on memory is represented by an inductive type `assign_loc` instead of `store_value_of_type`. As a note, since `CompCert` version supports volatile memory access, we also have to determine whether the object type is volatile before storage.

2. *Result of the evaluation of an expression with no modification on the memory.* Continuing the example above, we now discuss the result of evaluating the binary operation $Sbit\ ==\ 1$ both in the abstract and the concrete model. At the end of evaluation, a boolean value *true* or *false* is returned in both the concrete and the abstract models.

if `Sbit_related` M `Sbit`,
and $G, E \vdash$ `eval_rvaluebinop_`c $(Sbit\ == \ 1), M \Rightarrow\ v, M'$
then $v = (Sbit\ == \ 1)_{coq}$

Intuitively, the projection corresponding to the parameter `sbit` in the concrete model must yield the same value as in the abstract model. Here, the expression is a so-called simple expression that always terminates in a deterministic way, and preserves the memory state. To evaluate the value of simple expressions, `CompCert` provides two big-step relations `eval_simple_rvalue` and `eval_simple_lvalue` for Evaluating, respectively, their left and right values. The rules have the following shape:

$$\frac{G, E \vdash a_1, M \Rightarrow v_1 \quad G, E \vdash a_2, M \Rightarrow v_2 \qquad \texttt{sembinaryoperation}(op, v_1, v_2, M) = v}{G, E \vdash (a_1\ op\ a_2), M \Rightarrow v}$$

In order to evaluate the binary expression $a_1\ op\ a_2$, the subexpressions a_1 and a_2 are first evaluated, and their respective results v_1 and v_2 are used to compute the final result v.

3. *Memory state changed by storage operation or side effects.*
As mentioned before, evaluating some expressions such as `eval_assign` may modify the memory state. Lemmas are required to state that corresponding variables in the abstract and in the concrete model must evolve consistently. For example, considering an assignment on register Rn, the projection relation `register_related` is used. Expressions with side effects of modifying memory are very similar.

if `rn_related` $M\ rn$
and $G, E \vdash$ `eval_assign`$_c$ $(rn := rx), M\ \Rightarrow\ M', v$
then `rn_related` $M'\ rn$

4. *Internal function call.*
The simulation code is sometimes using functions from libraries. We distinguish `internal` functions and `external` functions. An internal function is a function that belongs to a library, the code of which is part of the simulator, that we have coded ourselves, or the C code is provided by compcert C. An external function is a function for which we do not have access to the operational semantics. After an internal function is called, a new stack of blocks is typically allocated in memory. After the evaluation of the function, these blocks will be freed. Unfortunately, this may not bring the memory back to the previous state: the memory contents may stay the same, but pointers and memory organization may have changed.

if `proc_state_related` $M\ st$
and $G, E \vdash$ `eval_funcall_`$(copyStatusRegister)\_c, M \Rightarrow v,\ M'$
and $st' = (copyStatusRegister)\_coq\ st$
then `proc_state_related` $M'\ st'$.

Lemmas must be developed regarding the evaluation of internal functions, so that one can observe the returned results, compare it with the corresponding evaluation in the formal specification, and verify some conditions. For example, the lemma above is about the processor state after evaluating an internal function call `copy_StatusRegister`, which reads the value of the Current Processor Status Register (CPSR) and copies it into the Saved Processor Status Register (SPSR) when an exception occurs. The evaluation of `copy_StatusRegister` must be protected by a check on the current processor mode. If it is in authorized mode, the function `copy_StatusRegister` can be called. Otherwise, the result is "unpredictable," which is defined by ARM architecture.

It is necessary to reason on the newly returned states, which should still be related by the projection. This step is usually easy to prove, by calculation on the two representations of the processor state to verify that they match.

5. *External function call.*
The `CompCert` C AST of an external function call contains the types of input arguments and of the returned value, and an empty body. For each external function (e.g., `memcpy()`), we have its asserted properties, mostly provided by `CompCert` C. The general expected properties of an external call are that (i) the call returns a result, which has to be related to the abstract state, (ii) the arguments must comply with the signature. (iii) after the call, no memory blocks are invalidated, (iv) the call does not increase the access permission of any valid block, and finally that the memory state can be modified only when the access permission of the call is granted. For each external call, such required properties are verified.

5.3.4.5 Inversion

Equipped with these lemmas we can build the proof scripts for ARM instructions. For that, we are decomposing the ARM instruction execution step by step to perform the execution of the C programs. `CompCert` C operational semantics define large and complex inductive relations. Each constructor describes the memory state transformation of an expression, statement, or function. As soon as we want to discover the relation between memory states before and after evaluating the C code, we have to *invert* the hypotheses of operational semantics to follow the clue given by its definition, to verify the hypotheses relating concrete memory states according to the operational semantics.

An *inversion* is a kind of forward reasoning step that allows for users to extract all useful information contained in a hypothesis. It is an analysis over the given hypothesis according to its specific arguments, that removes absurd cases, introduces relevant premises in the environment and performs suitable substitutions in the whole goal. Most proof assistants provide an inversion mechanism. In the case of Coq, it is a general tactic called `inversion` [10].

Every instruction contains complex expressions, but each use of `inversion` will go one step only. If we want to find the relation between the memory states affected by these expressions, we have to invert many times. For illustration, let us consider the simple example from the ARM reference manual `CPSR = SPSR`, that assigns to register CPSR the value of SPSR (defined above). As the status register is not implemented by a single value, but a set of individual fields, the corresponding C code is a call to the function `copy_StatusRegister`, which sets the CPSR field by field with the values from SPSR. Lemma `same_cp_SR` below states that the C memory state of the simulator and the corresponding formal representation of ARM processor state evolve consistently during this assignment.

```
Lemma same_copy_SR :
  ∀ e m l b s t m′ v em,
  proc_state_related m e (Ok tt (mk_semstate l b s)) →
    eval_expression (Genv.globalenv prog_adc) e m expr_cp_SR t m′ v  →
    ∀ l b, proc_state_related m′ e
                      (Ok tt (mk_semstate l b (Arm6_State.set_cpsr s
                                               (Arm6_State.spsr s em))))
```

In its proof, 18 consecutive inversions are needed in order to exhaust all constructors occurring in the assumptions. Unfortunately, `inversion` generates uncontrollable names which pollute proof scripts. Here, an intensive use of `inversion` makes proofs scripts unmanageable, and not robust to version changes of Coq or `CompCert`. In order to reduce the script size and get better maintainability, we studied a general solution to the inversion problem, and developed a new mechanism described in [26]. On top of it, we could program a Coq tactic able to automatically find the hypothesis to invert by matching the targeted memory states, properly manage other hypotheses, perform our inversion, clean up the goal, and repeat the above steps until all transitions between the two targeted memory states are discovered.

As a result, proofs script have become much shorter and more manageable. Considering the former example of `same_copy_SR`, the 18 calls to standard `inversion` reduce into one single step: `inv_eval_expr m m′`.

5.3.4.6 Instruction Proofs

Proofs of instructions rely heavily on the library of lemmas and the controllable inversion mechanism described above. Scripts size vary with the instructions complexity from less than 200 lines (e.g., 170 for LDRB) to over 1000 (1204 for ADC). As a result, for each ARM instruction, we have established a theorem proving that the C code simulating an ARM instruction is equivalent to the formal specification of the ARM processor.

5.4 Conclusion

The SimSoC virtual prototyping software is a full system simulation framework. Based on SystemC and TLM, it contains fast instruction set simulators and a library of models for other hardware components such as RS232 controller, or network controllers. Thanks to TLM interfaces, these models can interact with other third party models to elaborate complete simulation platforms that can run an operating system starting from a bootloader.

`SimSoC` can fully simulate a complete hardware platform. In addition to the ISS, it also includes implementation of several Memory Management Units (MMU's), interrupt controllers, serial line, and network controllers. All of these simulation models are implemented as SystemC modules using transaction modeling. As a proof of concept, we have developed several simulators to simulate commercially available System-on-Chips. All of the SoC's models developed are running the Linux operating system, using the Linux binary as is for the commercial chip. In order to boot Linux, it is also possible to use a boot loader such as U-Boot. The SoCs available are

- the SPEArPlus 600 circuit from ST Microelectronics. This SoC contains among other components two ARM926 subsystems (dual core), together with various peripheral controllers.
- the Texas Instrument AM1705 circuit. For this SoC, we have also developed an Ethernet controller model, and a bridge to real Internet so that we can test the simulated platform connected to real machines on the network, thanks to a bridge with the local Ethernet port.
- the FreeScale 8641D dual core Power architecture chip.
- We have under construction an example of the Power ez200 series, for which we develop an Approximately Timed version.

In order to build faithful virtual prototypes, we have added to SimSoc a proven generated simulator of the ARM instruction set.

Using the approach presented here, we can construct a tool chain that makes it possible to certify that the simulation of a binary executable program on some simulation platform is compliant with the formal model of the target hardware architecture. Using Compcert-C, that has defined formal C semantics, we have formally proved, using the Coq theorem prover to automate the proof, the ARM v6 Instruction Set Simulator of SimSoc.

Given that we have a proof that the machine code generated from C is correct, thanks to `CompCert`, and now a proof of the ARM instruction set for these instructions, we have a proof that the simulation of an algorithm on our simulator is conforming to the algorithm for the target architecture. With this technique, there is no limit on the size of the C code that can be verified. In fact, if there existed a publicly available formal model of the ARM processor approved by ARM Ltd company, our work, combined with CompCert C compiler, could be construed to define a *verified execution of a C program*, that could be used for certification procedures.

We certainly acknowledge the limits of our approach: the quality of our "verified simulation" relies on the faithfulness of our formal model of the ARM processor to the real hardware. Because the vendor companies do not provide a formal description of their hardware, one has to build them.[1] This issue is partly solved in this work by automatically deriving the most tedious parts of the Coq formal model from pseudo-code extracted from the vendor reference manual. If the vendors would make public formal specifications of their architectures, then our toolchain would become fully verified.

We believe this work has further impact on proofs of programs. First, we have proved here a significantly large C program. Second, because the proved program is a hardware simulator, it can be used as a tool to prove execution of target programs. For example, considering a cryptographic algorithm implemented for the ARM architecture and compiled with Compcert-C, it could then be proved that the execution of that program provides the exact encryption required, and nothing else. Therefore, the tool presented is an enabler for the proofs of other programs, which offers a direction for future research.

Another consequence of this work is that, supposing one could compile the C instructions to silicon using a silicon compiler, and that compiler would also be certified, ala `CompCert`, it would then make it possible to prove real hardware, etc.

Acknowledgements This work has been partly funded by the international collaboration support of France ANR and NSFC China in the SIVES project.

References

1. J. Alglave, A. Fox, S. Ishtiaq, M.O. Myreen, S. Sarkar, P. Sewell, F.Z. Nardelli, The semantics of power and ARM multiprocessor machine code. In *DAMP'09* (ACM, New York, NY, USA, 2008), pp. 13–24
2. V. Bala, E. Duesterwald, S. Banerjia, Dynamo: a transparent dynamic optimization system. SIGPLAN Not. **35**(5), 1–12 (2000)
3. F. Bellard, Qemu, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the Annual Conference on USENIX Annual Technical Conference* (USENIX Association, Berkeley, CA, USA, 2005), pp. 41–41
4. Y. Bertot, P. Castéran, *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science (Springer, New York, 2004)
5. F. Bobot, J.C. Filliâtre, C. Marché, A. Paskevich, Why3: Shepherd your herd of provers. Boogie **2011**, 53–64 (2011)
6. D. Burger, T.M. Austin, The simplescalar tool set, version 2.0. SIGARCH Comput. Archit. News **25**(3), 13–25 (1997)
7. B. Campbell, An executable semantics for Compcert C. In *Certified Programs and Proofs* (Springer, New York, 2012), pp. 60–75

[1]Note that this problem is the same as for the work done by Cambridge University.

8. G. Canet, P. Cuoq, B. Monate, A value analysis for C programs. In *SCAM'09* (IEEE, Los Alamitos, 2009), pp. 123–124
9. B. Cmelik, D. Keppel, Shade: A fast instruction-set simulator for execution profiling. In *SIGMETRICS '94: Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (ACM, New York, NY, USA, 1994), pp. 128–137
10. Coq Development Team, *The Coq Reference Manual, Version 8.2*. INRIA Rocquencourt, France, 2008. http://coq.inria.fr/
11. J.-C. Filliâtre, C. Marché, The Why/Krakatoa/Caduceus platform for deductive program verification. In *Proceedings of the 19th International Conference on Computer Aided Verification, Lecture Notes in Computer Science 4590*, 2007. http://why.lri.fr/
12. A.C.J. Fox, M.O. Myreen, A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture. In *ITP*, pp. 243–258, 2010
13. Y. Futamura, Partial evaluation of computation process—an approach to a compiler-compiler. Higher Order Symb. Comput. **12**(4), 381–391 (1999)
14. F. Ghenassia (ed.), *Transaction-Level Modeling with SystemC. TLM Concepts and Applications for Embedded Systems* (Springer, New York, 2005). ISBN 0-387-26232-6
15. G. Hamerly, E. Perelman, B. Calder, How to use simpoint to pick simulation points. SIGMETRICS Perform. Eval. Rev. **31**(4), 25–30 (2004)
16. C. Helmstetter, V. Joloboff, SimSoC: A SystemC TLM integrated ISS for full system simulation. In *IEEE Asia Pacific Conference on Circuits and Systems - APCCAS'08*, November 2008. http://formes.asia/cms/software/simsoc
17. H. Hongwei, S. Jiajia, C. Helmstetter, V. Joloboff, Generation of executable representation for processor simulation with dynamic translation. In *Proceedings of the International Conference on Computer Science and Software Engineering* (IEEE, Wuhan, China, 2008)
18. IEEE, *Open SystemC Language Reference Manual*, 2011. http://standards.ieee.org/getieee/1666/download/1666-2011.pdf
19. V. Joloboff, X. Zhou, C. Helmstetter, X. Gao, Fast Instruction Set Simulation Using LLVM-based Dynamic Translation. In *International MultiConference of Engineers and Computer Scientists 2011*, vol. 2188, *IAENG* (Springer, Hong Kong, China, 2011), pp. 212–216
20. D. Jones, N. Topham, High speed cpu simulation using ltu dynamic binary translation. In *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers, HiPEAC '09* (Springer, Berlin, Heidelberg, 2009), pp. 50–64
21. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, S. Winwood, sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles, SOSP '09* (ACM, New York, NY, USA, 2009), pp. 207–220
22. C. Lattner, V. Adve, LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004
23. X. Leroy, Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009)
24. X. Leroy, *The CompCert C verified compiler. Documentation and user's manual*. INRIA Paris-Rocquencourt, March 2012. http://creativecommons.org/licenses/by-nc-sa/3.0/
25. W. Liu, M.C. Huang, Expert: expedited simulation exploiting program behavior repetition. In *Proceedings of the 18th Annual International Conference on Supercomputing, ICS '04* (ACM, New York, NY, USA, 2004), pp. 126–135
26. J.-F. Monin, X. Shi, Handcrafted inversions made operational on operational semantics. In *ITP 2013* vol. 7998 of *LNCS*, ed. by S. Blazy, C. Paulin, D. Pichardie (Springer, Rennes, France, 2013), pp. 338–353
27. A. Nohl, G. Braun, O. Schliebusch, R. Leupers, H. Meyr, A. Hoffmann, A universal technique for fast and flexible instruction-set architecture simulation. In *DAC '02: Proceedings of the 39th Conference on Design Automation*, DAC '02 (ACM, New York, NY, USA, 2002), pp. 22–27
28. Open SystemC Initiative, *OSCI SystemC TLM 2.0 User Manual*, 2008. http://www.systemc.org/

29. C. Pusch, Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL. In *TACAS'99* (Springer, New York, 1999), pp. 89–103
30. W. Qin, J. D'Errico, X. Zhu, A multiprocessing approach to accelerate retargetable and portable dynamic-compiled instruction-set simulation. In *CODES+ISSS'06* (ACM, New York, NY, USA, 2006), pp. 193–198
31. M. Reshadi, P. Mishra, N. Dutt, Instruction set compiled simulation: a technique for fast and flexible instruction set simulation. In *Design Automation Conference, 2003. Proceedings*, pp. 758–763, 2003
32. K. Scott, N. Kumar, S. Velusamy, B. Childers, J.W. Davidson, M.L. Soffa, Retargetable and reconfigurable software dynamic translation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'03)*, 2003
33. D. Seal, *ARM Architecture Reference Manual* (Addison-Wesley Longman Publishing, Boston, 2000)
34. H. Shi, Y. Wang, H. Guan, A. Liang, An intermediate language level optimization framework for dynamic binary translation. SIGPLAN Not. **42**(5), 3–9 (2007)
35. E. Witchel, M. Rosenblum, Embra: fast and flexible machine simulation. In *SIGMETRICS '96: Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (ACM, New York, NY, USA, 1996), pp. 68–79
36. R.E. Wunderlich, T.F. Wenisch, B. Falsafi, J.C. Hoe, Smarts: accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings. 30th Annual International Symposium on Computer Architecture, 2003*, pp. 84–95, 2003
37. Z. Zhang, V. Joloboff, X. Zhou, C. Helmstetter, Fast dynamic translation using llvm on multi-core hosts. In *5th Workshop on Architectural and Microarchitectural Support for Binary Translation (AMAS-BT)* (Intel Corporation, Portland, Oregon, USA, June 2012)

Chapter 6
A Composable and Predictable MPSoC Design Flow for Multiple Real-Time Applications

Seyed-Hosein Attarzadeh-Niaki, Ekrem Altinel, Martijn Koedam, Anca Molnos, Ingo Sander, and Kees Goossens

6.1 Introduction

Embedded system designers are required to integrate an increasing number of complex applications running on a single system on chip. This calls for design flows which first, start from abstract application models and implement them in a fully automated fashion; and second, support designing each application in isolation while preserving its behavior in the integrated system. The challenge becomes more harsh for embedded systems that have real-time constraints where the design flows and the target platforms also need to preserve the timing properties of each application throughout the entire design flow.

Code generator backends from model-based design tools for real-time processors have been proposed [14, 22] but complexities of a real-time MPSoC design flow are not fully addressed in these works. Existing design flows that perform real-time analysis for MPSoCs are presented in [7, 9, 16, 25] without expressing the applications and components in an executable formal model and hence, a potential bug in the specification will be detected late in the design flow and makes it

S.-H. Attarzadeh-Niaki (✉)
Shahid Beheshti University, Tehran, Iran
e-mail: h_attarzadeh@sbu.ac.ir

E. Altinel • I. Sander
KTH Royal Institute of Technology, Stockholm, Sweden
e-mail: altinel@kth.se; ingo@kth.se

M. Koedam • K. Goossens
Eindhoven University of Technology, Eindhoven, Netherlands
e-mail: M.L.P.J.Koedam@tue.nl; K.G.W.Goossens@tue.nl

A. Molnos
CEA-LETI, Grenoble, France
e-mail: Anca.MOLNOS@cea.fr

A. Molnos, C. Fabre (eds.), *Model-Implementation Fidelity in Cyber Physical System Design*, DOI 10.1007/978-3-319-47307-9_6

also harder to track it in the original specification. Additionally, running multiple applications on the same platform is not supported in these works.

We argue that proper design flows for real-time systems require to

(a) start from abstract application models with formal semantics that are executable to enable detection of specification bugs early in the design flow, avoiding long design iterations;
(b) apply *automated analysis and synthesis* methods supported by the applications' formalisms; and
(c) target platforms which provide *time-predictable* execution services to faithfully implement the application behavior; and support *composable* system design by integrating isolated applications.

Building on a common formal base in form of Models of Computation (MoCs), we introduce such a flow for multiple real-time streaming applications which integrates **For**mal **Sy**stem **De**sign (ForSyDe), as a modeling and simulation framework, with CompSOC, a design flow and platform template for predictable MPSoCs (Fig. 6.1). However, our methods are applicable to any combination of system-level design languages and target platforms which satisfy the above requirements. In the SystemC implementation of ForSyDe [20] a formally defined representation of the executable system models can be exported as an intermediate format. CompSOC is a network-on-chip based MPSoC platform template which provides predictable and composable execution services to the applications. The platform currently has a design flow which performs automated mapping, compilation, and synthesis

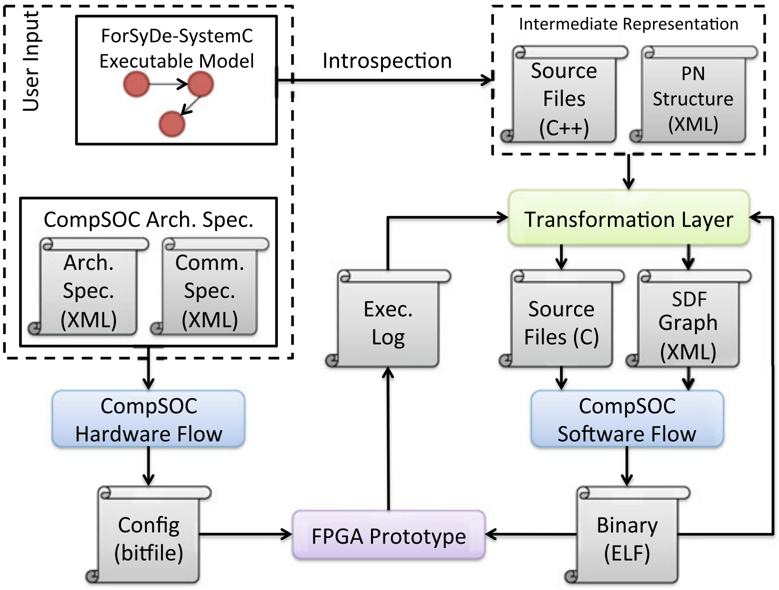


Fig. 6.1 The proposed system design flow

from a non-validatable implementation model, where the structure (as XML files) and behavior of the application (as a set of C files) are captured separately. By iterative transformation of exported ForSyDe-SystemC models, we integrate them with the CompSOC design flow and demonstrate an automated flow starting from high-level simulate-able formal application models. Such a transformation involves annotating platform-specific memory and timing requirements of the application elements which is obtained by rapid performance evaluation of the application on the platform. This results a correct-by-construction design flow which preserves the functionality and timing of application models in the final implementation.

The contributions of this work are summarized as:

- an automated and composable design flow implementing abstract executable models of multiple applications on predictable platforms (Sect. 6.3);
- transformation of the specification models captured in the formal modeling framework (Sect. 6.4) to the implementation models accepted by the predictable platform design flow (Sect. 6.5) which involves rapid performance evaluation of the application models on the platform (Sect. 6.6);
- demonstration of the flow in action using two applications from the consumer electronics domain (Sect. 6.7).

6.2 Related Work

Several tools and design flows have been proposed for real-time MPSoCs, but none of them fully address different aspects of the problem.

The industrial tool Simulink can produce plain C code from executable real-time models. In [10], a model-based design flow for cyber-physical systems is presented in ten design steps, but it is not fully automated. These approaches bring interesting ideas in the field, however they do not consider real-time constraints throughout the entire design-flow.

The PTIDES flow [3] targets event-based applications described in a Programming Model with discrete-event semantics. First the temporal behavior and causality relations of the application model are analyzed and then actor and OS code are generated. This code may be compiled and bound to three hardware platforms, out of which one, namely the XMOS board, provides real-time services. The flow simulates the application code bound on the platform to verify and validate the design. The effects of the binding on the temporal behavior are hence not formally analyzed, which may lead to run-time constraints violations.

Code generator backends for real-time processors has been proposed for executable Ptolemy [5] models, notably for JOP [22], which is a Java real-time processor, and PRET [14] which aims at providing predictability in its architecture. However, complexities of a real-time MPSoC design flow are not fully addressed in these works.

Existing design flows that perform real-time analysis for MPSoCs are presented in [7, 16, 25]. The approaches in [16, 25] target the streaming domain and have as input an XML description of the application. The flow produces the code of the application, and the information necessary to bind this application on a predictable platform. Real-time constraints are guaranteed, as the temporal behavior of application, the binding and the platform are formally verified. Note that due to compatibility of the Synchronous Data Flow (SDF) graph formats, the SDF^3 tool [25] can be replaced for formal temporal analysis in our flow. The SymTA/S [7] framework models an SoC as a set of inter-connected components. Each component is modeled by an event stream. Classical formal real-time analysis can be applied to individual components and a formalism for composing streams is proposed to analyze an entire system. The distributed operation layer (DOL) MPSoC software design flow [9] targets dataflow real-time streaming applications and automatically creates an implementation as well as formal performance analysis models for system validation. However in these approaches applications and components are not expressed in an executable formal model, hence a potential bug in the specification will be detected late in the design flow and makes it also harder to track it in the original specification.

Deaedalus^{RT} methodology [2] starts with a Static Affine Nested Loop Program (SANLP) and uses the hard real-time multiprocessor scheduling techniques to analyze the system from intermediate Cyclo-Static Data Flow (CSDF) models and performs code generation via Polyhedral Process Networks (PPNs). In contrast, being based on the ForSyDe framework, our input models have explicit parallelism and are easier to extend and interact with other MoCs for a heterogeneous system design. On the platform side, CompSOC can run multiple applications based on resource reservation composably, reducing the run-time scheduling overhead for admission control. Also, our flow ends with a full FPGA-based prototype while Daedalus-based flows have reported simulation backends.

The method proposed by Posadas et al. [21] presents an automated software synthesis flow from a UML/MARTE specification containing system components, interfaces and communication links, the system memory spaces, the resource allocations, and the hardware architecture. A software stack is then generated from such a specification automatically to execute the application on the platform. Unlike our approach, automatic system-level Design Space Exploration (DSE) based on formal models is not considered and preservation of real-time properties is not guaranteed throughout the design flow.

6.3 The Proposed Design Flow

In the high-level view of the proposed flow depicted in Fig. 6.1, the designer uses the SDF MoC library of ForSyDe-SystemC to create formal executable system models. This model is used both for validation by simulation and also for automated generation of an intermediate representation of the specification model as a set of XML and C++ files via introspection. Since both ForSyDe and CompSOC are based on formally defined MoCs, they can be combined by transforming the specification model to the target implementation model to achieve a correct-by-construction flow. The benefit of basing the models on formally defined abstractions is evident in the fact that orchestrating the modeling framework (ForSyDe), a dataflow analysis tool, and the synthesis flow (CompSOC) in our case is straightforward and does not involve any semantical ad-hoc transformations. We assume the hardware platform is given as an architecture description and a communication description file and it is not going to be explored in the design flow. To hand over ForSyDe-SystemC models to the CompSOC flow, an automated transformation step is needed to iteratively add the missing platform information to the model (Sect. 6.6). Figure 6.2 details build iterations of the software flow which are presented as a loop in Fig. 6.1.

The proposed flow supports multiple applications and runs in three stages:

1. *initial build*, which performs the model transformation assuming constant values for platform-dependent metrics, performs a single core mapping and scheduling, and generates a binary file for each application;
2. *measurement run*, where the generated binaries and source files are analyzed to extract the actor memory requirements and token sizes and applications are run to measure the execution times of the actors; and

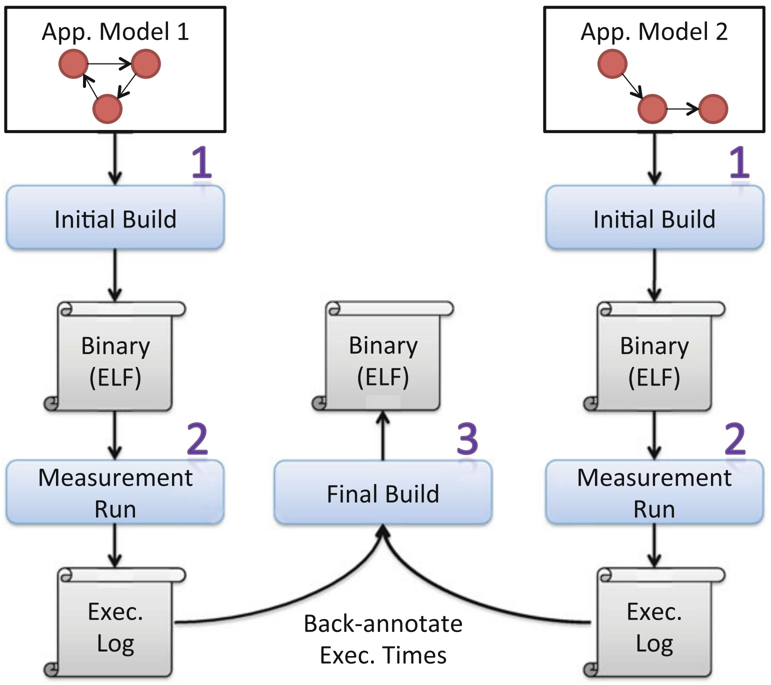


Fig. 6.2 Build iterations of the proposed software flow

3. *final build*, in which the SDF graphs of all applications with all the platform-dependent metrics back-annotated are merged, the final mapping and scheduling of the combined dataflow graph is performed, and software synthesis is done to generate the final binaries.

The first two phases are performed individually for each applications, since CompSOC preserves composability [1].

The CompSOC platform is predictable and composable, as explained in more detail later. The important thing to note at this point is that each application runs independently of others: there is not a single cycle of interference due to other applications for every run. (This holds for actual-case timings as well as worst-case timings.) Moreover, the performance of each platform component is predictable, i.e. the worst case can be computed at design time. As a result, the execution time of each actor (e.g., software task) can be measured independently of all other actors in the measurement run, even when they are mapped on the same processor (in the initial build). This execution time does not change if we remap the actor on another (identical) processor (in the final build). We can do this per application, as the execution time does not change when adding other applications, due to composability. In theory it is possible to run the final build per application rather than all applications together, but then the resulting configurations (virtual platforms) are not co-optimized.

6.4 The Modeling Framework

In ForSyDe-SystemC [20], a system model is structured as a hierarchical concurrent process network. Processes communicate and synchronize *only* using signals and there is no global state in the system. Hierarchy does not imply any semantics and is simply a grouping of multiple processes; however, it enables easier IP reuse. Figure 6.3 is an example of a system model in which p_1 is a composite process formed by composition of leaf processes $p_{1.1}$, $p_{1.2}$, and $p_{1.3}$.

Each leaf process in the process network belongs to a specific Model of Computation (MoC) [13]. Several MoCs are supported for system modeling in ForSyDe. This work addresses the synthesis of SDF models [12] which fits very well to many streaming applications and is supported by analysis methods for consistency checking, temporal scheduling, and mapping to single- and multi-processor systems.

Leaf processes are created using formally defined constructs chosen from the ForSyDe library called *process constructor* which are provided with side-effect-free functions and/or initial values. In Fig. 6.3, a process is created using the *combSDF* process constructor which is supplied with the firing function f and two initial values representing the production and consumption rates r_p and r_c to gain a process with the semantics of an actor in the SDF MoC. By using the concept of process constructors the requirements for the computation and communication semantics of

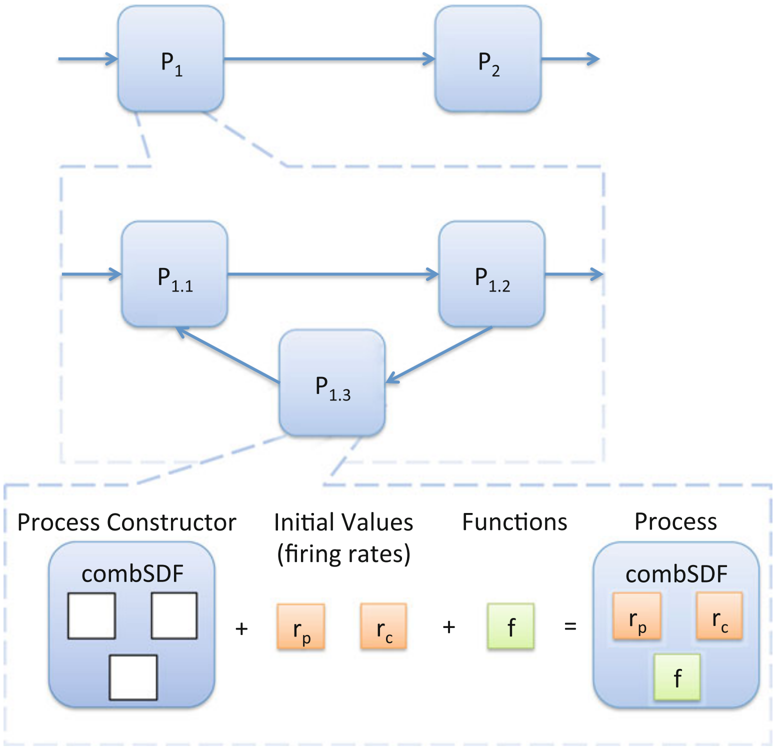


Fig. 6.3 Example of a hierarchical system model in the SDF MoC of ForSyDe. Leaf processes are built using process constructors

the processes are satisfied by construction and the designer is liberated from writing redundant code and focuses on the pure functionality of the processes.

There are two key process constructors in the SDF MoC;

1. $combSDF_m$, which denotes an m-input combinational (i.e., stateless) process constructor in the SDF MoC; and
2. $delaySDF_n$ which delays a signal by n elements.

Because processes in ForSyDe are formally defined as mathematical functions operating on input signals and returning a single output signal, tuples of values are used to model multiple signals. Such a signal can be converted into multiple signals using special family of processes called *unzip* and created using *zip* processes.

A special feature of ForSyDe-SystemC models is that in addition to simulation, the constructed executable models can export their internal structure and behavior as an intermediate representation via *introspection*. The exported representation can be used to feed the models to analysis and synthesis tools, without developing a full-fledged compiler infrastructure. This intermediate representation contains the structure of the process network, the process constructors used to build leaf processes, and the parameters passed to build the processes. These parameters include both the initial values and the source code of the functions which describes the behavior of the processes [20].

6.5 The Execution Platform

CompSOC [6] provides predictable execution to the applications and can run multiple applications without interference in a composable manner [1]. Time-division multiplexing (TDM) is used to provide time-predictable execution, communication, and memory access services while composability is achieved by using arbiters which prevent indefinite locking of shared resources.

The hardware is a collection of processor and memory tiles interconnected by a dAElite NoC [24] (Fig. 6.4). The NoC consists of protocol shells, which serialize the parallel protocol, network interfaces (NIs), which (de-)packetize the information and inject/collect them to/from the network in a TDM fashion, and routers. The processor tiles include a Microblaze softcore without caches, local instruction and data memories, cycle-accurate times, a set of communication memory blocks and direct memory access (DMA) engines for inter-tile communication, and other peripherals. Memory tiles are divided into two parts, namely front-end and back-end. The front-end contains a number of blocks to achieve composability while the back-end guarantees the predictability of the resource [1].

CompSOC can run a composable RTOS which uses one or two-level scheduling for applications. The inter-application scheduler is a microkernel named CoMik [8, 18], which uses the TDM technique to provide individual performance guarantees to multiple integrated applications. The timers in the tile are hooked up to

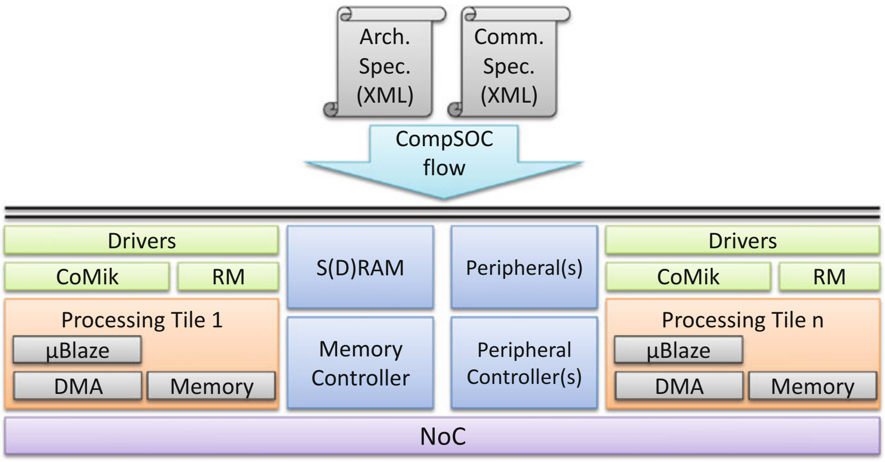


Fig. 6.4 Overview of the CompSOC platform and its generation flow

the processor interrupts to support cycle-accurate accounting of time, scheduling of interrupts at a deadline, and halting the processor until a deadline. The intra-application scheduler (task scheduler) supports executing application tasks with different semantics, namely KPN, CSDF, and also time-triggered MoCs [15].

The local memory of each tile stores the instructions and data of each task executing on that tile. The communication memories store all input (i.e., SDF input tokens) that the actor requires, and reserves space for all output (i.e., SDF output tokens) that the actor produces in one firing. As a result, an actor never stalls on input or output during its execution.

Predictability [6] is achieved by making sure that every resource has a finite known execution time for an actor that is mapped on it. For example, data transport from one port on the Network on Chip (NoC) to another is modelled by a communication actor, and the worst-case execution time of that actor can be computed at design time. The execution time of DMAs, DRAM, and SRAM is similarly bounded, taking into account the configuration of their arbiters (e.g., number of time slots). Similarly, as long as no DMAs are used to communicate, the execution time of software running on a processor depends only on the processor speed, since the software does not stall on caches or other communication outside the tile. Adding a microkernel or RTOS with one-level scheduling adds a "slow-down factor" (essentially, the total number of TDM slots divided by the number of allocated TDM slots) [19]. The second level of arbitration can be dealt with in the same way [17].

To achieve a composable system, each shared resource of CompSOC is composable [6]. The execution of different applications must be independent, i.e. their actual (not: worst-case) execution times are independent. This implies that the order and duration of time allocated to applications is independent, and that resource arbiters are not work conserving between applications.

The supporting CompSOC design flow for SDF applications consists of three sub-flows, hardware generation, mapping and software compilation flow. The hardware generation flow takes in the communication and architecture models of the platform and performs dimensioning, allocation, verification, instantiation, and synthesis of the hardware architecture. Based on the SDF graph of the application and the architecture model of the platform, the mapping flow invokes a mapping tool to explore the design space and generates a mapping and schedule of the application onto the platform, and finally synthesizes the software for each core. Consecutively, the compilation flow is invoked for each tile in the platform and the generated binary ELF file is merged with the platform bit-stream file generated by the hardware generation flow.

6.6 Adapting the Flows by Rapid Performance Evaluation

Recall that our approach has three phases. First, an initial build consisting of transforming the input application model to the SDF graph supported by the software synthesis flow. The result is then compiled on a single processor. This compilation results in code size (of actors) and data size (including sizes of data tokens). Note that remapping and recompiling actors and data FIFOs on different (but identical) resources will not change these sizes. This first step is non-trivial, and we give more details below.

Second, in the measurement run we execute the SDF graph on the single processor. Recall that processors have no caches, and that any communication outside the tile requires the use of DMAs. This, coupled with the SDF semantics actor has all its input tokens when it fires as well as space for the tokens that it outputs, means that the execution time of an actor only depends on the processor it runs on. The execution time of an actor obtained on a given processor is therefore identical to its execution time on another (identical) processor. A complication is that the actor may share the processor with different applications. However, since the Comik microkernel and the RTOS provide cycle-accurate isolation between applications and also cycle-accurate timing, we can remove the influence of other applications on the actor timing. Similarly, we can remove timing influence of sharing with other actors of the same application in a static-order schedule.

Finally, in the final build, we remap actors and FIFOs to multiple processors such that any real-time constraints are met. As mentioned above, we can do this, without changing code and data sizes, and execution times. The predictable and composable performance of our platform, in particular the processor tile, is therefore an essential enabler for our three-phased approach.

To enable full automation, the pure SDF input model is transformed for synthesis. The execution times and memory requirements of the actors and also the token sizes are estimated using a rapid estimation technique. Auxiliary processes in ForSyDe process networks are transformed to their equivalent SDF actors. Additionally, the interface and API of the functions are different and conversion between them is required. Figure 6.5 is a simplified view of the transformation stage between the flows.

Figures 6.6 and 6.7 demonstrate a small ForSyDe model in the SDF MoC and its equivalent model transformed automatically by the proposed flow for the CompSOC backend. This synthetic example is not a common situation in system modeling, but illustrates different aspects of the transformation processes.

Transforming ForSyDe process networks to SDF graphs involves

(a) flattening their static non-recursive hierarchy;
(b) removing *zip*, *unzip*, and *fanout* processes by integrating them with multi-input/output actors;
(c) converting *source* and *constant* state-full processes to actors with self-edges; and
(d) converting delay elements to initial tokens on the graph edges.

Figure 6.6a shows the structure of a process network as described in the SDF MoC of ForSyDe, while Fig. 6.7a demonstrates how it should be exported to the CompSOC backend as an SDF graph. As stated in Sect. 6.4, hierarchy eases component reuse and *zip*/*unzip* processes are used to combine/extract multiple

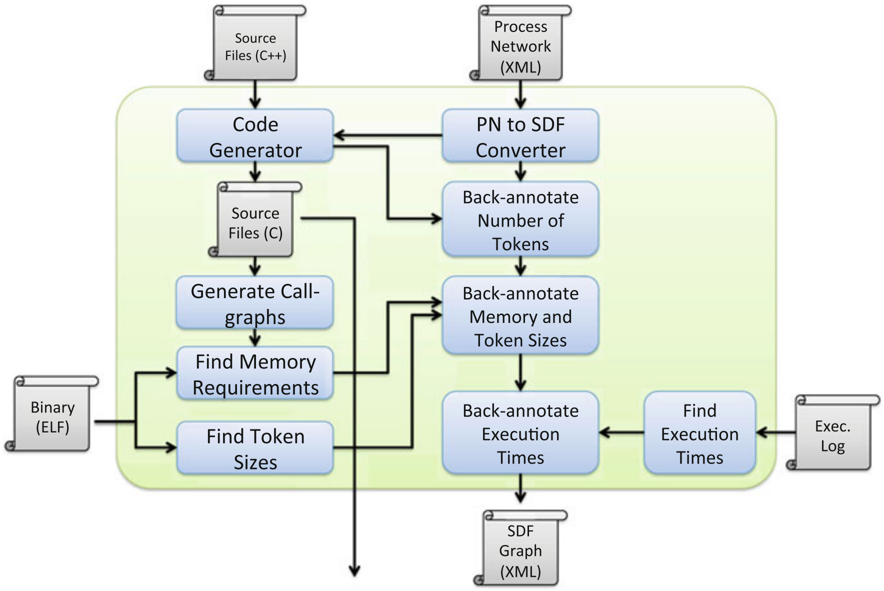


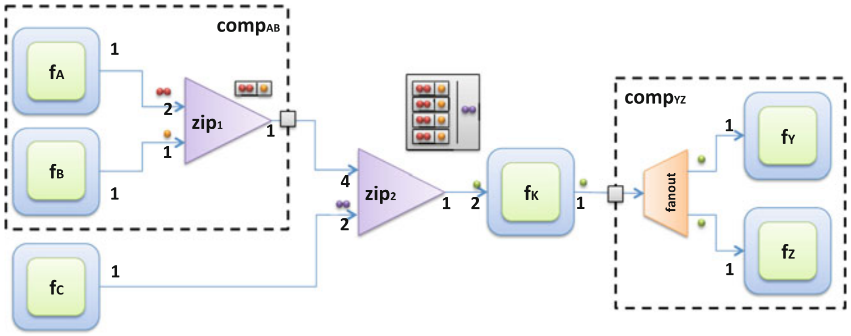
Fig. 6.5 The transformation layer between the flows

Algorithm 1 Determination of the consumption and production rates

```
$a2alinks$ ←all actor-to-actor links
for all $link \in a2alinks$ do
    $zipfactor \leftarrow multiply$(zip rates in $link$)
    $unzipfactor \leftarrow multiply$(unzip rates in $link$)
    $channelfactor \leftarrow zipfactor$ div $unzipfactor$
    if $channelfactor > 1$ then
        $targetrate \leftarrow targetrate * channelfactor$
    else if $channelfactor < 1$ then
        $sourcerate \leftarrow sourcerate * (1/channelfactor)$
    else if both actors' ports are tuples then
        Report error: Unbalanced zip and unzip sequence
    else
        Keep the original source and target rates
    end if
end for
```

signals to/from a signal of tuples. Both $comp_{AB}$ and $comp_{YZ}$ composite processes are eliminated in the flattening stage. While removing *zip*s and *unzip*s, all actor-to-actor paths are considered and the production and consumption rates are transformed using a traversal algorithm which is presented as pseudo-code in Algorithm 1. Conversions are realized using XSL transformations [11].

The generated SDF graph needs to be annotated with resource requirements and timing information before the CompSOC backend can make a mapping satisfying the constraints. This is done in two phases, first a single tile mapping of the a



b

```
 void k_func(std::vector<float> &out1,
   const std::vector<std::tuple<std::vector
     <std::tuple<std::vector<float>, std::vector<int>>>,
     std::vector<UserType>>> &inp1,
   const std::vector<UserType> &inp2) {

   // A macro interface for the signal flattener utility
   FLATTEN_INPUTS(inp1, inp2);
   FLATTEN_OUTPUTS(out1);

#pragma ForSyDe begin k_func // Beginning of the extractable code block

   // Inputs are read using I(S, N),
   // S : the index of the channel
   // N : index of the token in the buffer
   // Outputs are written by assigning to O(S, N)

   O(0, 0) = process_1(I(0, 0), I(1, 0), I(2, 0), I(3, 0));

#pragma ForSyDe end // End of the code block
}
```

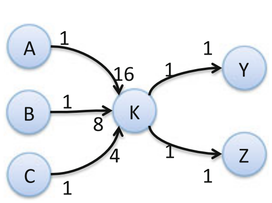
Fig. 6.6 A (synthetic) model in the SDF MoC of ForSyDe. (**a**) The process network. (**b**) The code for function f_x

application is generated and compiled for the target platform. During this process the following information is gathered;

(a) *Token sizes* The DWARF [4] output data generated from running the GNU `readelf` tool on the binary files is analyzed to extract the size of each data token.
(b) *Actor memory requirements* The memory requirement of each individual function is retrieved using the GNU tool `nm`, this combined with the call-graphs created using the LLVM Clang tool chain is analyzed to recursively find the memory requirements for each of the SDF actor.

This information is required to make a valid mapping of the application on the platform and this information is back annotated into the SDF graph. In the second phase the application is mapped using this extra information, compiled and executed. By executing the application on the platform an execution log is obtained.

a



b

```
void k_func() {
  FLATTEN_INPUTS();
  FLATTEN_OUTPUTS();

  INPUT_VAR(0, float);
  INPUT_VAR(1, int);
  INPUT_VAR(2, UserType);
  INPUT_VAR(3, UserType);
  OUTPUT_VAR(0, float);
  OUTPUT_VAR(1, float);

  // Inputs are read using I(S, N),
  // S : the index of the channel
  // N : index of the token in the buffer
  // Outputs are written by assigning to O(S, N)

  O(0, 0) = process_1(I(0, 0), I(1, 0), I(2, 0), I(3, 0));

  // Fanouts
  for (int __k = 0; __k < 1; __k++) {
    O(1, __k) = O(0, __k);
  }
}
```

Fig. 6.7 The automatically transformed model of Fig. 6.6 for mapping to CompSOC. (**a**) The SDF graph. (**b**) The code for function f_x

This execution log is analyzed to get an estimation of the worst case execution time for each SDF actor. This information is then added to the SDF graph, that now contains all the information required to make a real-time mapping.

As for adapting the code for individual actor functions, the main logic is copied without modification but the function interface is adapted for the backend. In the above example, Fig. 6.6b shows how process `k` is modeled in ForSyDe-SystemC, while Fig. 6.7b demonstrates its equivalent generated code for the SDF actor `k`. In both cases, the `process_1` function is the fixed part of the logic. The initial ForSyDe-SystemC model uses standard C++ `vector` and `tuple` data structures for the function inputs and outputs which are transformed by the library-provided macros before being used by the main logic. On the other hand, the generated code uses functions from the CompSOC library to prepare input and output data structures accessible to the function logic.

Note that the above sketched process is simplified; generating code for the SDF actors not only needs transformation between the models and APIs for the two flows, but also requires extracting and matching information such as the initial token values from the ForSyDe intermediate representation, generating code for *constant* processes, and generating the required header functions and initialization code.

A single tile mapping of the application for determining execution times might not be possible due to resource limitations. In this case, the tools will automatically map to the minimum number of tiles but this does not affect the strategy. Additionally, if the application uses dynamic memory allocation, extra measurement runs might be needed to determine the minimum heap size.

6.7 Case Study

To demonstrate the feasibility of the proposed design flow, we apply it to two applications from the multimedia domain, namely SUSAN edge detection and JPEG decoder. The experiments are conducted to examine the correctness of the design flow in terms of preserving the functionality and timing characteristics of multiple high-level application models in the final implementation. We demonstrate that a quick single-core mapping of the application onto the platform is sufficient for measuring an execution time bound for the application actors. These execution time bounds can be used in the final build of the design flow considering multiple applications at the same time.

Smallest Univalue Segment Assimilating Nucleus (SUSAN) [23] includes three signal processing algorithms, out of which the edge detection algorithm is considered here. The image is partitioned into smaller blocks and the following steps are applied to each pixel:

1. a mask is applied to an area (called USAN) centered around each pixel of interest;
2. the direction of the edge is detected by calculating the momentums of the USAN area; and
3. thinning is applied on the edges to clarify the pixels.

JPEG standard [26] is a commonly used lossy compression method for images. In this case study we consider a JPEG decoder. The decoding of an image in this experiment is performed in five steps by:

1. parsing the image headers and decompressing the input as a series of 8×8 pixel blocks;
2. inverse quantization and reordering of blocks;
3. combining pixel blocks into RGB pixel values; and
4. putting the pixel values in the final image.

The CompSOC platform instance used for this case study has three tiles: a shared memory tile with 256 KB of memory and two processor tiles running at 120 MHz. Each processor tile contains 256 KB of instruction and data memory and three sets of communication memories and DMAs. The tiles are connected by an NoC with two routers and four NIs. The platform is prototyped on a Xilinx ML-605 Virtex 6 FPGA board. The platform is setup to generate, without affecting the execution of the application, a detailed trace of the execution.

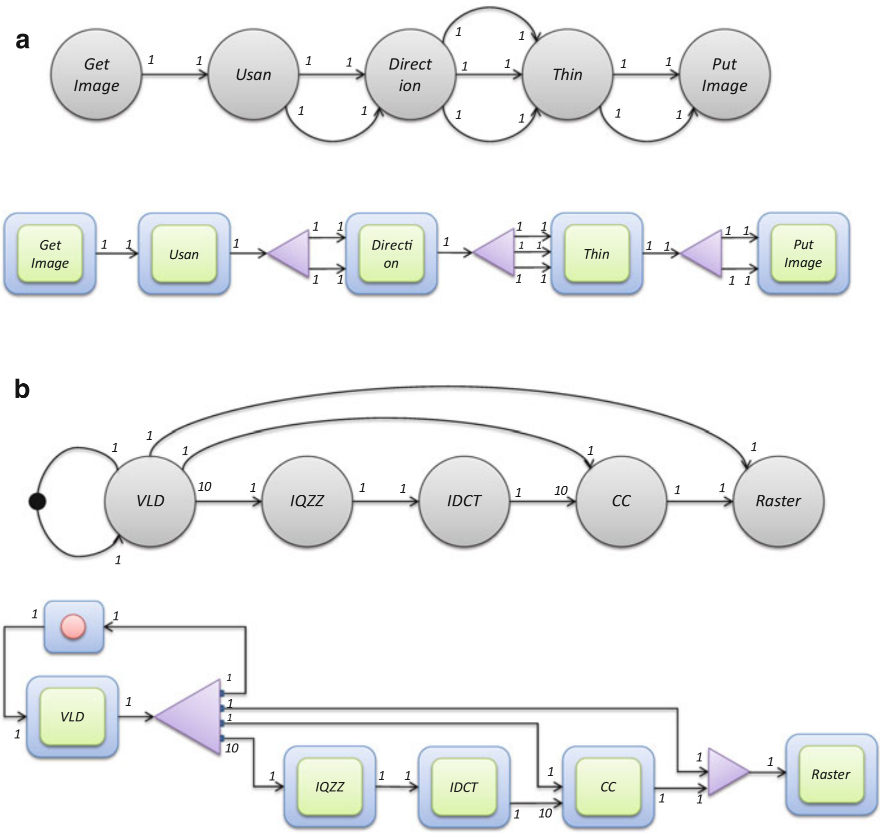


Fig. 6.8 The ForSyDe process network and the SDF graph of the application models. (**a**) SUSAN. (**b**) JPEG decoder

First, a model of each applications is developed using the SDF MoC of ForSyDe. Figure 6.8a, b illustrate their process networks as captured by ForSyDe-SystemC together with the result of their transformation into an SDF graph. These models are verified by simulation, with the same testbench that is used later in the measurement run, by checking the produced output against a reference considered correct.

The design flow is automated using makefiles and can be invoked from the command line using a single make command. The first two main stages of the design flow, namely the initial build and the measurement run stages are executed for the two applications separately. The applications are separately run on the platform to collect the execution times of the actors. The results obtained by the measurement runs, maximum measured execution times of each actor, are presented in Table 6.1.

Table 6.1 Execution times of the actors and the mapped application on the platform (in clock cycles)

SUSAN	getImage	USAN	Direction	Thin	putImage	App.
	20077	1177105	833912	35843	15866	1356352
JPEG	VLD	IQZZ	IDCT	CC	Raster	App.
	626884	4294	15505	21284	1327	61618352

Table 6.2 Execution times of different parts of the flow (in seconds)

	Simulate and introspect	Individual flow	Combined flow
SUSAN	0.16	549	1056
JPEG	0.01	509	

After the back annotation of the execution times, the applications are mapped one after each other on the same platform, resulting in final mapping containing both applications. Table 6.2 summarizes the execution times of each part of the flow.[1]

The final phase which is executed for both applications (1056 s) is less than half of the total execution time of the flow (2114 s). Note that we have used a simple greedy mapper in our experiments. In a more elaborate design-space exploration tool, the individual builds for measurement runs would be a smaller fraction of the final build. After producing the bit-stream file, the system is run and both of the applications are verified to produce the correct output. Table 6.1 includes the execution time of individual applications in the final implementation. These times remain constant while the applications run individually and both together on the platform.

6.8 Conclusion and Future Work

We have proposed a fully automated design flow for multiple real-time signal processing applications which compiles formal executable specifications to a predictable MPSoC template. The design flow

(a) moves the design entry to a higher level of abstraction since functional models can be simulated efficiently in SystemC;
(b) provides an automated path to synthesis using the introspection feature of ForSyDe and the CompSoC tool suite; and
(c) uses rapid performance estimation of the applications on the target platform to estimate platform-specific metrics of the applications.

[1]Experiments are run on a 64 bit Linux machine with a Core i7 CPU running at 3.07 GHz with 24 Gb of memory.

Only half of the execution time of the flow is consumed for the combined application model since the platform ensures the composability of the individually analyzed applications.

We plan to enrich the flow by supporting additional MoCs such as the Synchronous (SY) MoC for control-oriented behavior, integrating a more elaborate dataflow analysis tool such as SDF^3, and also aiming at a mixed-criticality design flow. Also, the method we have used for rapid estimation of the actor execution times can be enhanced by employing static worst-case execution time analysis tools.

References

1. B. Akesson, A. Molnos, A. Hansson, J. Angelo, K. Goossens, Composability and predictability for independent application development,verification, and execution, in *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*, ed. by M. Hübner, J. Becker (Springer, New York, 2011), pp. 25–56
2. M.A. Bamakhrama, J.T. Zhai, H. Nikolov, T. Stefanov, A methodology for automated design of hard-real-time embedded streaming systems, in *Proceedings of the Conference on Design, Automation and Test in Europe. DATE '12* (EDA Consortium, San Jose, 2012), pp. 941–946
3. P. Derler, J. Eidson, S. Goose, E. Lee, S. Matic, M. Zimmer, Using Ptides and synchronized clocks to design distributed systems with deterministic system wide timing, in *Symposium on Precision Clock Synchronization for Measurement Control and Communication (ISPCS)* (2013), pp. 41–46
4. M.J. Eager, E. Consulting, Introduction to the DWARF debugging format (2007), http://www.dwarfstd.org
5. J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, Y. Xiong, Taming heterogeneity - the Ptolemy approach. Proc. IEEE **91**(1), 127–144 (2003)
6. K. Goossens, A. Azevedo, K. Chandrasekar, M.D. Gomony, S. Goossens, M. Koedam, Y. Li, D. Mirzoyan, A. Molnos, A.B. Nejad, A. Nelson, S. Sinha, Virtual execution platforms for mixed-time-criticality systems: the CompSOC architecture and design flow. ACM Spec. Interest Group Embed. Syst. Rev. **10**(3), 23–34 (2013). http://doi.acm.org/10.1145/2544350.2544353
7. A. Hamann, M. Jersak, K. Richter, R. Ernst, A framework for modular analysis and exploration of heterogeneous embedded systems. Real-Time Syst. **3**, 101–137 (2006)
8. A. Hansson, M. Ekerhult, A. Molnos, A. Milutinovic, A. Nelson, J. Ambrose, K. Goossens, Design and implementation of an operating system for composable processor sharing. Microprocess. Microsyst. **35**(2), 246–260 (2011). Special issue on Network-on-Chip Architectures and Design Methodologies
9. K. Huang, W. Haid, I. Bacivarov, M. Keller, L. Thiele, Embedding formal performance analysis into the design cycle of MPSoCs for real-time streaming applications. ACM Trans. Embed. Comput. Syst. **11**(1), 8:1–8:23 (2012)
10. J. Jensen, D. Chang, E. Lee, A model-based design methodology for cyber-physical systems, in *Proceedings of the 7th International Conference on Wireless Communications and Mobile Computing (IWCMC)* (2011), pp. 1666–1671
11. M. Kay, et al., XSL transformations (XSLT) version 2.0. W3C Recom. (2007)
12. E. Lee, D. Messerschmitt, Synchronous data flow. Proc. IEEE **75**(9), 1235–1245 (1987)
13. E. Lee, A. Sangiovanni-Vincentelli, A framework for comparing models of computation. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **17**(12), 1217–1229 (1998)

14. B. Lickly, I. Liu, S. Kim, H.D. Patel, S.A. Edwards, E.A. Lee, Predictable programming on a precision timed architecture, in *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems. CASES '08* (ACM, New York, 2008), pp. 137–146
15. A. Molnos, A.B. Nejad, B.T. Nguyen, S. Cotofana, K. Goossens, Decoupled inter- and intra-application scheduling for composable and robust embedded MPSoC platforms, in *Proceedings of the 15th International Workshop on Software and Compilers for Embedded Systems. SCOPES '12* (ACM, New York, 2012), pp. 13–21
16. O. Moreira, Temporal analysis and scheduling of hard real-time radios running on a multi-processor. Ph.D. thesis, Technical University of Eindhoven (2012)
17. A.B. Nejad, A. Molnos, K. Goossens, A software-based technique enabling composable hierarchical preemptive scheduling for time-triggered applications, in *Proc. Int'l Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)* (2013)
18. A. Nelson, A.B. Nejad, A. Molnos, M. Koedam, K. Goossens, CoMik: a predictable and cycle-accurately composable real-time microkernel, in *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE)* (2014)
19. A. Nelson, K. Goossens, B. Akesson, Dataflow formalisation of real-time streaming applications on a composable and predictable multi-processor SOC. J. Syst. Archit. **61**, 435–448 (2015)
20. S.A. Niaki, M. Jakobsen, T. Sulonen, I. Sander, Formal heterogeneous system modeling with SystemC, in *Proceedings of the Forum on Specification and Design Languages (FDL)* (2012), pp. 160–167
21. H. Posadas, P. Peñil, A. Nicolás, E. Villar, Automatic synthesis of embedded SW for evaluating physical implementation alternatives from UML/MARTE models supporting memory space separation. Microelectron. J. **45**(10), 1281–1291 (2014). http://www.sciencedirect.com/science/article/pii/S0026269213002607. DCIS'12 Special Issue
22. M. Schoeberl, C. Brooks, E. Lee, Code generation for embedded Java with Ptolemy, in *Software Technologies for Embedded and Ubiquitous Systems*, ed. by S. Min, R. Pettit, P. Puschner, T. Ungerer. Lecture Notes in Computer Science, vol. 6399 (Springer, Berlin/Heidelberg, 2010), pp. 155–166
23. S. Smith, J. Brady, SUSAN–a new approach to low level image processing. Int. J. Comput. Vis. **23**(1), 45–78 (1997)
24. R. Stefan, A. Molnos, K. Goossens, dAElite: a TDM NoC supporting QoS, multicast, and fast connection set-up. IEEE Trans. Comput. **99**, 1–10 (2012)
25. S. Stuijk, M. Geilen, T. Basten, SDF3: SDF for free, in *Proceedings of the Sixth International Conference on Application of Concurrency to System Design (ACSD)* (2006), pp. 276–278
26. G.K. Wallace, The jpeg still picture compression standard. Commun. ACM **34**(4), 30–44 (1991). http://doi.acm.org/10.1145/103085.103089

Chapter 7
Analysis and Implementation of Embedded System Models: Example of Tags in Item Management Application

Bojan Nokovic and Emil Sekerinski

7.1 Introduction

Verification of probabilistic systems is a technique for establishing if quantitative properties hold for a particular system model. The properties are expressed in temporal logic extended with probabilistic and reward operators. The model can be specified by engineers in a high-level modelling language as a variant of Markov chain processes annotated with costs and rewards, and used as input for a probabilistic model checker, e.g. [1]. Such system models can serve only for analysis.

Traditional state machines are flat and sequential in nature. To effectively allow representing complex behavior, such as that of communication protocols, statecharts, which are hierarchical state machines with concurrency and broadcasting were introduced [2]. Hierarchy is a structuring method that allows the developer to maintain an overview of large and complex applications. The most abstract view is at the outermost level and zooming in reveals details in lower level views. The design process begins with an outline of the application and then stepwise adds functionality. Concurrency and broadcasting are used to describe parallel tasks and communication.

Statecharts are used as a graphical specification tool for reactive systems, but they are executable and compilable like programming languages [3]; pCharts extend statecharts further with probabilistic transitions, timed transitions, stochastic timing, state invariants, and costs/rewards assigned to states and transitions [4, 5]. pCharts

B. Nokovic (✉) • E. Sekerinski
Computing and Software Department, McMaster University, Main Street West, 1280, Hamilton, ON, Canada
e-mail: nokovib@mcmaster.ca; emil@mcmaster.ca

A. Molnos, C. Fabre (eds.), *Model-Implementation Fidelity in Cyber Physical System Design*, DOI 10.1007/978-3-319-47307-9_7

are supported by *pState*,[1] a tool for the *holistic* design: in addition to generating executable code, *pState* can be used to model the system's environment and to verify quantitative properties like resource consumption (e.g. power), reliability (e.g. lost messages, life expectancy), and performance (e.g. throughput). Such queries can be specified directly on pCharts.

We use an application with electronic tags to illustrate the holistic design process. Section 7.2 reviews the design process. Section 7.3 gives an overview of the architecture and functionality of *pState*. Section 7.4 describes the process of executable code generation from pCharts. Section 7.5 presents the process of generating Markov decision processes and probabilistic timed automata for the PRISM probabilistic model checker. Section 7.6 describes the generation of the executable code framework. The remaining sections present a case study: on the example of the DASH-7 ISO/IEC 18000-7.2 communication protocol, we first give a system collision model in Sect. 7.8, then a model of power consumption in Sect. 7.9, and finally the executable code framework in Sect. 7.10. The final section summarizes the contribution.

7.2 A Holistic Design Process

Existing automated tools for analyzing discrete, timed, probabilistic, or stochastic models have a textual user interface, which makes them less suitable for engineers developing larger systems. Visualization of models in the form of hierarchical state machines, like statecharts, allows a view where the whole system is represented from the perspective of related states. An extension of statecharts with probabilistic transitions, timed transitions, and stochastic timing is proposed in [6]. Invariantcharts, statecharts with state invariants are introduced in [7]. pCharts support probabilistic transitions, timed transitions, stochastic timing, state invariants and add costs/rewards assigned to states or transitions. Through the *pState* editor, the pChart system model is entered. Quantitative queries are specified directly in the pCharts. After validation, a system without timed transition can be verified over a Markov decision process for systems (MDP) and a system with timed transitions over a probabilistic timed automaton (PTA); these are passed to a probabilistic model checker. The correctness of transitions with respect to state invariants is checked with a combination of the probabilistic model checker and a satisfiability modulo theories (SMT) solver. Executable code for the software part of the system can be generated and its worst-case execution time (WCET) analyzed. The architecture of *pState* is in Fig. 7.1.

[1]http://pstate.mcmaster.ca.

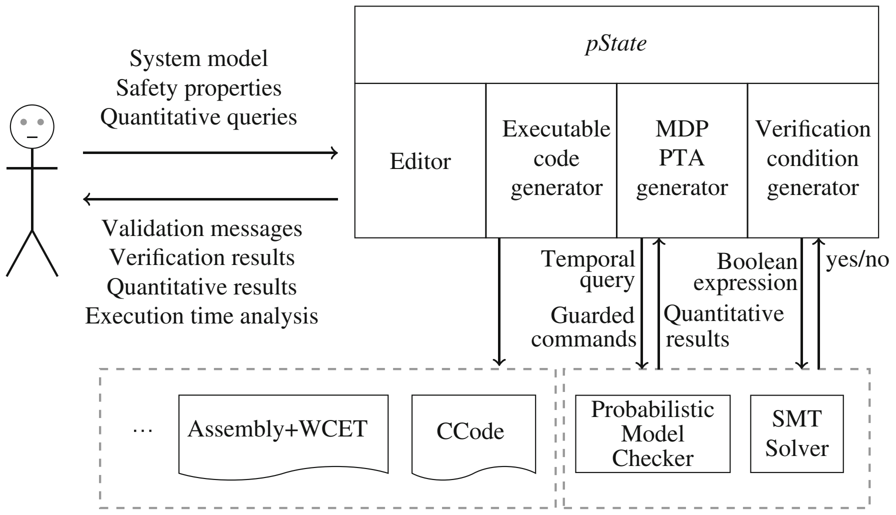
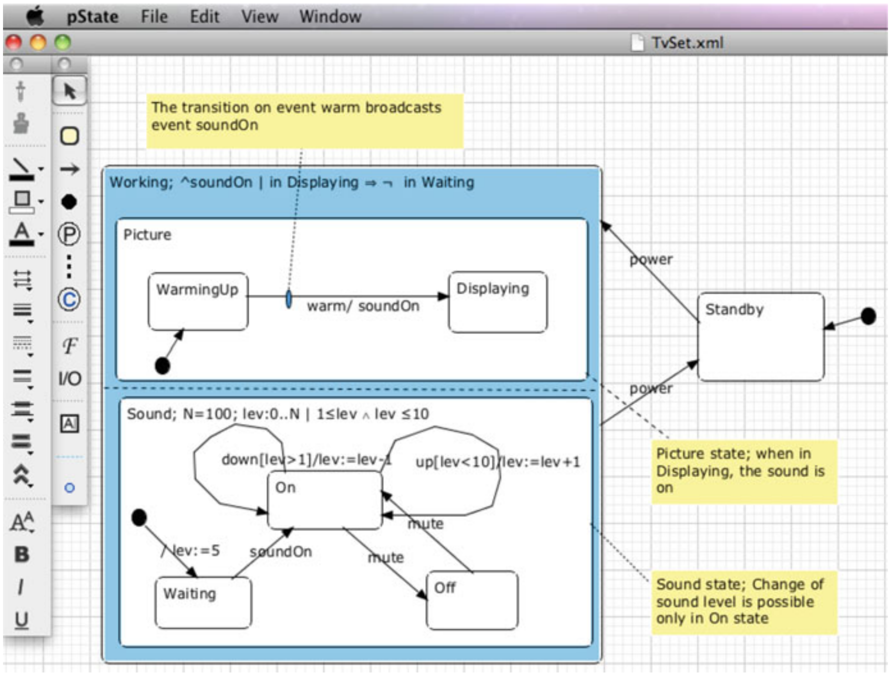


Fig. 7.1 Top-level *pState* architecture

7.3 pState Editor

The editor is designed on the JHotDraw (JHD) 7.6 framework [8]. As a starting point we use frameworks from the *org.jhotdraw.samples* package. Figure 7.2 is a view of the *pState* graphical interface, which shows features of a TV set represented as a pChart. Components like states and transitions are added in a drag-and-drop fashion using icons in the toolbar. States without children are called Basic states. On the TV set, chart states *Standby, WarmingUp, Displaying, Waiting, On*, and *Off* are Basic states. Compositional states are either AND states or XOR states. State *Working* is an AND state, it has two children, *Pictures* and *Sound*, separated by a dashed line. When the chart is in *Working*, it is at the same time in both *Picture* and *Sound*. Composite XOR states are (1) *Picture* with two Basic states *WarmingUp* and *Display*, (2) *Sound* with three children *Waiting, On*, and *Off*, and (3) the top state *root* with two children, *Working* and *Standby*.

The TV control activity is partitioned into two states, the Basic state *Standby*, and AND state *Working*. The initial state is *Standby*. When the chart is in *Working*, it is in both the *Picture* and *Sound* XOR states. Within *Picture* the chart is in one of the basic states *WarmingUp* or *Displaying*, within *Sound* the system is in one of the Basic states *Waiting*, *On*, or *Off*. The invariant of *Working* specifies that whenever *Picture* is in *Displaying*, *Sound* must not be in *Waiting*, i.e. must be either in *On* or *Off*. The invariant of *Sound* specifies that the sound level *lev* must be between 1 and 10; the invariant must be established by the initialization of *Sound* and be preserved by all transitions within *Sound*. The event *power* causes the chart to flip between *Standby* and *Working*, no matter in which substates of *Working* the chart is. The transition on event *warm* broadcasts event *soundOn*. The transition on events *down*



The transition on event warm broadcasts event soundOn

Picture state; when in Displaying, the sound is on

Sound state; Change of sound level is possible only in On state

Fig. 7.2 Statecharts with invariants for TV set

can only be taken if *lev* $>$ 1 and when taken, will decrement *lev*. The transition on *power* to *Working* sets *Picture* and *Sound* to the default initial states *WarmingUp* and *Waiting* and sets *lev* to 5.

The design tools in Fig. 7.2 are for selection, state (basic and composed), transitions, initial pseudostate, probabilistic pseudostate, concurrency line, choice pseudostate, quantitative query, and comment. It is straightforward to add other tools to the button factory. We additionally use the standard attribute bar with all selections from the JHD framework. This bar is an example of how new features, like colour of the figure, can be added to the drawing editor.

7.4 From Hierarchical Charts to Code

pState generates code according to an *event-centric* interpretation, in which events are executable procedures, implying that an event is processed before the next one arrives. This interpretation is according to the *requirements-oriented* semantics [9]. This is in contrast to the *implementation-oriented* semantics based on the *state-centric* interpretation in UML and Statemate [10], in which events are data in queues. The event-centric interpretation was already used by *iState*, the predecessor of *pState* [11]. The event-centric approach is suitable for those kind of reactive systems where events are processed quickly enough that queueing is not needed and where blocking of events is undesirable. This semantic is close to [12]. Currently we do not support *spontaneous* transitions—transitions without an event.

Hierarchical state machine diagrams consist, essentially, of just three components: a set of states, an initial state, and a set of transitions. The system starts at the initial state, then follows transitions on external events to move to other states. States can hold entire sub-state-machines within themselves. Concurrent states express orthogonality or independence. A transition t from a set of source states ss (of distinct concurrent states) to a set of target states tt (of distinct concurrent states), is a tuple written as $t = ss \xrightarrow{E[g]/b\ \$c} tt$, where E is the transition event, g is a Boolean expression, the transition guard, $\$c$ is a non-negative number, the cost of the transition and b is a statement, the transition body. In a *regular* transition, E is the event name, while in a *timed* transition E is the number of time units [5]. Transitions can be *probabilistic*, in which case target states are indicated as probabilistic alternatives [4]. Each transition must have E, while guard g, cost c, and body b are optional. All states are nested in the state *root*, which must not be the source or target of any transition.

In the transformation of a pChart to intermediate code, for each event, code associated with that event is generated and for every XOR state, an enumeration variable is generated holding the names of children states. The code generation is based on the recursive algorithm of [5].

The *scope* of a transition is the innermost state which contains all its source and target states. The grammar of the generated intermediate code has two mutually recursive productions, *Scopeop* and *Childop*. In the intermediate code, one variable for each state in the hierarchy is declared, starting with *root*, representing in which child state the system is. The algorithm for generating the *operation Op* of a regular event visits all transitions of one scope, starting with *root* as scope, before visiting transitions in children. The transitions on one scope are of the form

$$Trigger \rightarrow Effect \ [] \ \ldots \ [] \ Trigger \rightarrow Effect$$

with a *nondeterministic choice* ([]) among them, and each choice being *guarded* ($\rightarrow$). These transitions take *priority* (//) over transitions in children. If the child is an XOR state, there is first a test to determine in which state the system is (Test), followed by the transitions with that child as scope. If the child is an AND state, then transitions on that event in all children are taken in parallel ($\|$). The *trigger* of a transitions contains tests for all the source states of the transition ($Variable = State$) and the guard (*Expr*). The *effect* of a transition is executing the body of the transition (*Statement*) in parallel with moving to target states (*Goto*), with a probabilistic choice ($\oplus$) among such alternatives, such that the probabilities for each alternative ($Probability : \ldots$) sum up to 1. Thus the intermediate representation *Op* of a regular pCharts event is of the following form:

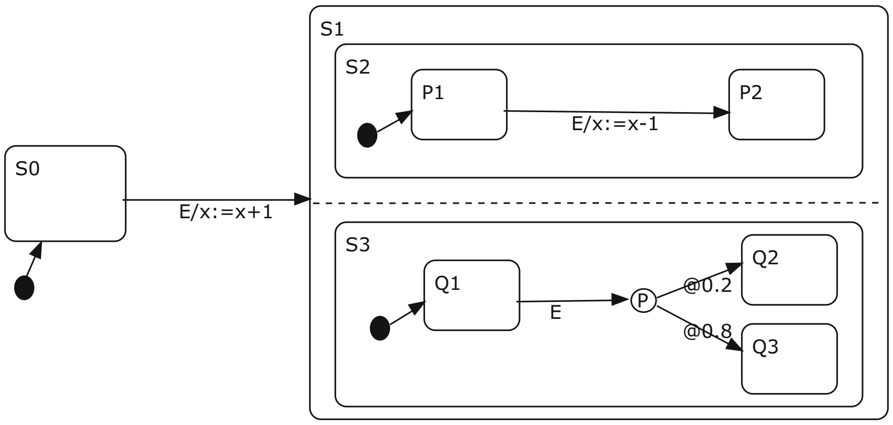


Fig. 7.3 Operation on event E

$$\begin{array}{lll}
\textit{Op} & ::= & \textit{Scopeop} \\
\textit{Scopeop} & ::= & (\textit{Trigger} \to \textit{Effect} \mathbin{[]} \cdots \mathbin{[]} \textit{Trigger} \to \textit{Effect}) \mathbin{/\!/} \textit{Childop} \\
\textit{Childop} & ::= & (\textit{Test} \to \textit{Scopeop} \mathbin{[]} \cdots \mathbin{[]} \textit{Test} \to \textit{Scopeop}) \mathbin{/\!/} \textbf{skip} \\
 & | & \textit{Scopeop} \parallel \cdots \parallel \textit{Scopeop} \\
\textit{Test} & ::= & \textit{Variable} = \textit{State} \\
\textit{Trigger} & ::= & \textit{Variable} = \textit{State} \wedge \cdots \wedge \textit{Variable} = \textit{State} \wedge \textit{Expr} \\
\textit{Effect} & ::= & \textit{Probability} : \textit{Statement} \parallel \textit{Goto} \oplus \cdots \oplus \textit{Probability} : \textit{Statement} \parallel \textit{Goto} \\
\textit{Goto} & ::= & \textit{Variable} := \textit{State} \parallel \cdots \parallel \textit{Variable} := \textit{State}
\end{array}$$

As an example, the operation $op(E)$ of the event E in Fig. 7.3 is as follows:

$$\begin{array}{l}
op(E) = \\
\qquad (root = S0 \to x := x + 1 \parallel root := S1 \parallel s2 := P1 \parallel s3 := Q1) \\
\qquad /\!/ \\
\qquad\quad root = S1 \to \\
\qquad\qquad (s2 = P1 \to x := x - 1 \parallel s2 := P2) \mathbin{/\!/} \textbf{skip} \\
\qquad\qquad \parallel \\
\qquad\qquad (s3 = Q1 \to 0.2 : s3 := Q2 \;\oplus\; 0.8 : s3 := Q3) \mathbin{/\!/} \textbf{skip} \\
\qquad\quad /\!/ \\
\qquad\qquad \textbf{skip}
\end{array}$$

The full description of *generalized program statements* **skip**, **stop**, multiple assignment, guarded statement, nondeterministic choice, probabilistic choice, and parallel composition used to define the meaning of events is in [5, 13]. The body of a transition is an action or *chart statement*, like $x := x + 1$. The grammar of chart statement is

$$
\begin{array}{lll}
\mathit{ChartStatement} & ::= & \textbf{if}\ \mathit{Expr}\ \textbf{then}\ \mathit{ChartStatement}\ [\textbf{else}\ \mathit{ChartStatement}]\ \mid \\
 & & \mathit{ChartStatement} \parallel \cdots \parallel \mathit{ChartStatement}\ \mid \\
 & & \mathit{Variable},\ldots,\mathit{Variable} := \mathit{Expr},\ldots,\mathit{Expr}\ \mid \\
 & & \mathit{Event} \\
\mathit{Expr} & ::= & \mathit{Variable} \mid \mathit{real} \mid \mathit{integer} \mid \textbf{true} \mid \textbf{false} \mid \mathit{UnOp}\ \mathit{Expr}\ \mid \\
 & & \mathit{Expr}\ \mathit{BinOp}\ \mathit{Expr}\ || \ \textbf{in}\ \mathit{State} \\
\mathit{UnOp} & ::= & - \mid \neg \\
\mathit{BinOp} & ::= & + \mid - \mid * \mid \textbf{div} \mid \textbf{mod} \mid = \mid \neq \mid < \mid \leq \mid > \mid \geq \mid \textbf{and} \mid \textbf{or}
\end{array}
$$

If an event leads to broadcasting of another event, the second one is executed in parallel with the first one, which imposes that there are no race conditions in the parallel execution. The translation of parallel statements needs extra processing since for executable code generation, parallel statements have to be converted into sequential statements using auxiliary variables. Parallel composition is first verified to be well defined such that variables assigned in parallel statements are disjoint, and then transformed to multiple assignments using the fact that $(x, y := E, F) = (x := E \parallel y := F)$ [5].

Before code generation, the *validation* performs three checks on charts: (1) Composite states must not be childless, AND state must have at least two children, each child of an AND state must be an XOR state; (2) all XOR states have initial transitions; (3) transitions between concurrent states are not allowed.

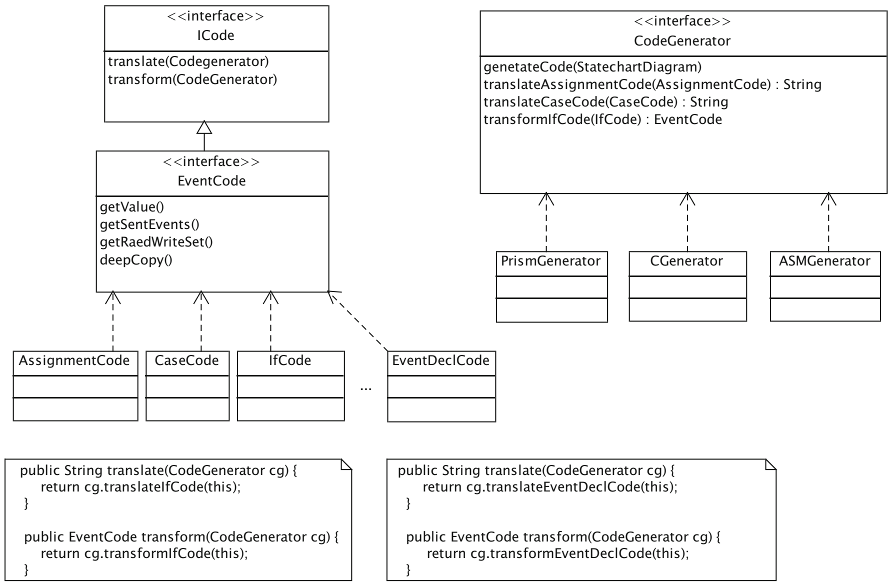
For target code generation, the *visitor* [14] pattern with two methods, *transform* and *translate* is employed, see Fig. 7.4. The elimination of parallel composition is done by *transform* and the creation of either executable code (C, assembly) or input code for a probabilistic model checker by *translate*.

7.5 Model Checker Input Code

From pCharts without timed transitions, *pState* generates an MDP model, and from pCharts with timed transitions, *pState* generates a PTA model as input for the PRISM model checker [15]. As PRISM requires the model to be a flat set of guarded commands with multiple (probabilistic) assignments as commands, after the elimination of parallel composition, the intermediate code is flattened. The full code generation algorithm is given in [5].

7.5.1 MDP

Markov decision processes are a variant of Markov chains that permit both probabilistic and nondeterministic choices. Our presentation of MDP follows [16–18].



```
public String translate(CodeGenerator cg) {
    return cg.translateIfCode(this);
}

public EventCode transform(CodeGenerator cg) {
    return cg.transformIfCode(this);
}
```

```
public String translate(CodeGenerator cg) {
    return cg.translateEventDeclCode(this);
}

public EventCode transform(CodeGenerator cg) {
    return cg.transformEventDeclCode(this);
}
```

Fig. 7.4 Class diagram of the visitor pattern in *pState*

Definition 1. A labelled Markov decision process is a tuple $M = (S, \bar{s}, A, p, l, r)$ where

- S is countable nonempty set of states;
- $\bar{s}$ is the set of initial states;
- A is the finite set of actions;
- p : S × A → Dist(S) is the transition probability function;
- l : S → 2^{AP} is the labelling function;
- r : S × A × S → R is the reward function.

and *AP* is a set of atomic propositions. We assume that *M* is time homogeneous; *S*, *A*, *p*, *l*, and *r* do not vary over time, and that *S* and *A* are discrete.

Example. The pChart of a simple MDP and the generated PRISM code are shown in Figs. 7.5 and 7.6. There are two transitions on *wakeup* from *S*0, the initial state, the choice between them being nondeterministic. One of the transitions is probabilistic, in which with 70 % probability state *S*1 is reached and with 30 % probability the system stays in the initial state. The other transition from *S*0 to *S*1 is deterministic, where on the event *wakeup* state *S*1 is always reached. The transition on the event *send* is deterministic and the transition on event *recv* is probabilistic. Rewards are assigned to the states by $\$r = e$, where and $e \geq 0$ is a real expression.

In *S*3 we specify two queries, the query ?$*r.max* returns maximum *costs* to reach *S*3, and the query ?*P.max* returns maximum *probability* to reach state *S*3. Those two properties are translated into PCTL formulae $R``r''max = ?[F(root = S3)]$ and

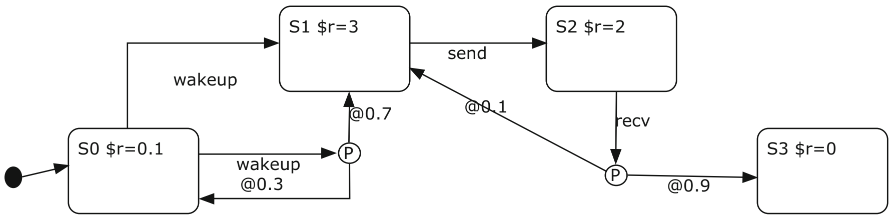


Fig. 7.5 State-transition diagram of the MDP model

```
mdp

const S0=0; const S1=1; const S2=2; const S3=3;

module mdpexample
      root :[0..3]   init  S0;

      [send]  (root=S1) −> (root'=S2);
      [wakeup] (root=S0) −> 0.3:(root'=S0) + 0.7:( root'=S1);
      [wakeup] (root=S0) −> (root'=S1);
      [recv]  (root=S2) −> 0.1:(root'=S1) + 0.9:( root'=S3);
endmodule

rewards "r"
      (root=S0):  0.1;
      (root=S1):  3;
      (root=S2):  2;
      (root=S3):  0;
endrewards
```

Fig. 7.6 MDP PRISM code generated by *pState*

$Pmin = ?[F(root = S3)]$, respectively. The calculated maximum costs to reach state $S3$ is 5.6984, and the maximum probability to reach $S3$ is 0.9999, that is 1. The error comes from floating point rounding of the model checker. In this example, there is a nondeterministic choice between the probabilistic and deterministic *wakeup* transitions from state S0 to state S1. Eventually, in both cases, the transition on event *wakeup* leads to S1 state. Calculated reward of 5.698 is maximum expected long-run reward. This is the same as a long-run average reward, but only if there are no nondeterministic transitions.

7.5.2 PTA

Timed automata (TA) provide a natural way for expressing timing delays of real-time systems [19]. On a TA, we can prove the correctness of finite-state real-time systems using the *trace* semantics originally proposed in a model for communicating sequential processes (CSP) [20]. Probabilistic timed automata (PTA) are an extension of TA used for formal modelling and analysis capabilities for systems with probabilistic, nondeterministic, and real-time characteristics [17]. PTA augmented with quantitative information in the form of costs or reward are called priced probabilistic timed automata. On a PTA model two main classes of properties can be analyzed, the minimum/maximum probability of reaching a target, possibly within a time bound and the minimum/maximum expected reward accumulated until a target is reached, using *quantitative abstraction refinement* and *statistical model checking* verification methods [21].

Definition 2. A probabilistic timed automaton (PTA) is a tuple
$P = (S, \bar{s}, \mathfrak{X}, A, inv, enab, prob, l)$, where

- S is the countable nonempty set of states;
- $\bar{s}$ is the set of initial states;
- $\mathfrak{X}$ is a finite set of clocks;
- A is the finite set of actions;
- inv : S $\rightarrow$ CC($\mathfrak{X}$) is an invariant condition, a clock constraint for each state;
- enab: S $\times$ A $\rightarrow$ CC($\mathfrak{X}$) is an enabling condition;
- prob: S $\times$ A $\rightarrow$ Dist($2^{\mathfrak{X}} \times$ S) is a (partial) probabilistic transition function;
- l : S $\rightarrow$ 2^{AP} is the labelling function;

Example. The pChart of a simple PTA and the generated PRISM code are shown in Figs. 7.7 and 7.8. The PTA has clock *rootclk* with initial value 0. In the state $S0$, the system waits for the *wakeup* event for 1 time unit. State $S0$ also allows a transition to state $S1$ when $rootclk = 1$ and the PRISM invariant $root = S0 \Rightarrow rootclk \leq 1$ forces the transition to be taken when *rootclk* reaches 1. In the state $S0$, the system waits for *wakeup* for a maximum of 1 time unit. If the event does not occur, it goes to next state on timed transition. On this model, properties like the expected time to reach state $S3$ or the probability of reaching state $S3$ in a given number of time units can be verified. In the state $S3$, we specify two queries: $?P.maxF < 10s$ returns 0.9, the maximum probability to reach state $S3$ in 10 time units (seconds) and $?P.minF < 10s$ returns 0.819, the minimum probability to reach $S3$ in 10 time units. Those properties cannot be verified on an MDP model. While PRISM uses abstract time units, in pCharts the time unit, here s, must be explicitly specified.

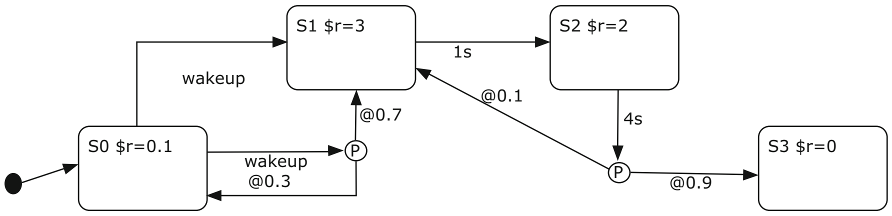


Fig. 7.7 State-transition diagram of the PTA model

```
pta

const S0=0; const S1=1; const S2=2; const S3=3;

module ptaexample
    root :[0..3]  init S0;     rootclk : clock;

    invariant
        (root=S1=>rootclk<=1)& (root=S2=>rootclk<=4)
    endinvariant

    [wakeup] (root=S0) −> (root'=S1)&(rootclk'=0);
    [wakeup] (root=S0) −> 0.3:(root'=S0)&(rootclk'=0) + 0.7:( root'=S1)&(rootclk'=0);
    []  (root=S2)&(rootclk=4) −> 0.1:(root'=S1)&(rootclk'=0) +
                                             0.9:( root'=S3)&(rootclk'=0);
    []  (root=S1)&(rootclk=1) −> (root'=S2)&(rootclk'=0);
endmodule

rewards "r"
    (root=S0): 0.1;
    (root=S1): 3;
    (root=S2): 2;
    (root=S3): 0;
endrewards
```

Fig. 7.8 PTA PRISM code generated by *pState*

A PTA in PRISM is verified by one of two engines, *digital clocks* [22] and *stochastic games* [23]. The specification of queries or *quantitative properties* of a PTA is based on probabilistic computational tree logic PCTL [17, 24]. In the digital clock engine, clock variables are allowed in P (probability) operator expressions, as well as in F (*eventually*) and U (*until*) expressions. However, this engine does not support time-bounded reachability properties and clock constraints cannot use strict comparison operators, e.g. $rootclk < 2$. Also, comparison between clock variables is not allowed. Automata with such constraints are called *closed*, *diagonal-free* probabilistic timed automata. The digital clocks method is based on a language-level translation from a PTA model to an MDP model. In the *stochastic games* engine, properties cannot contain references to clocks. Only unbounded or time-bounded probabilistic reachability properties are allowed. For this, only the P operator is used. The basic types of path properties that can be used inside the P operator are: X (*next*), U (until), F (eventually), G (*always*), W (*weak until*), and R (*release*), but the *stochastic game* engine currently (V 4.2.1) only supports the F path operator. The S operator, used to reason about the steady-state behavior of model, and the R operator, used to calculate reward properties, are not supported.

7.5.3 Properties Specification

pState allows quantitative queries to be placed inside hierarchical states, making use of the state hierarchy, while the specification of properties in PRISM is done separately from the model.

For example, in *pState* we can attach $?P.min$ to a state, say S, to compute the minimal probability to reach S. If S is child of *root*, *pState* generates the PCTL formula $Pmin\ =?[F(root\ =\ S)]$ for PRISM. The same query can be attached to another state, possibly deeper in the hierarchy, and *pState* would generate a corresponding, more complex property specification. Similarly, if the reward property $?\$tran.max$ for computing the maximal reward to reach that state is placed in S, *pState* generates $R\{``tran''\}max\ =?[F(root\ =\ S)]$. Quantitative queries in *pState* are according to the following grammar:

$$\begin{array}{ll}
\mathit{Query} & ::= ?(\mathit{Probability} \mid \mathit{Reward})(\mathbf{.min} \mid \mathbf{.max} \mid > \mathit{real} \mid < \mathit{real})[\mathit{Bound}][\mathit{Target}] \\
\mathit{Bound} & ::= \mathbf{F} < \ \mathit{Time} \\
\mathit{Target} & ::= '('\mathit{Expr}')' \\
\mathit{Probability} & ::= \mathbf{P} \\
\mathit{Reward} & ::= \$\mathit{Identifier} \\
\mathit{Time} & ::= \mathit{digit}\{\mathit{digit}\}(\mathbf{d} \mid \mathbf{h} \mid \mathbf{s} \mid \mathbf{ms}) \\
\mathit{Identifier} & ::= \mathit{letter}\{\mathit{letter} \mid \mathit{digit}\}
\end{array}$$

Quantitative queries are attached to a state or written in the special property box. Currently only simple properties can be attached to states. For more complex properties, which include more than one condition, property boxes have to be used. For instance, the PCTL formula $Pmax\ =?[F(rootclk\ <\ T)\&(root\ =\ S)]$ has to be specified in a property box. With this property we can calculate the probability that state S will be reached before T time units.

7.6 Executable Code

Target code is created by further translating the intermediate code, provided that there are no probabilistic transitions in the sub-chart for which code is to be generated. The intermediate code may contain parallel compositions emerging from broadcasting (transitions in concurrent states are taken in parallel) and multiple assignments. As multiple assignments are a special case of parallel composition, both are treated uniformly by introducing auxiliary variables and sequentializing, for example:

$$(x := y || y := x) = (x, y := y, x) = (\mathbf{var}\, h = x; x := y; y := h)$$

Specifications of costs/rewards are ignored for code generation. Nondeterministic choice with guarded choices is translated as **if-then-else** or **case** statements in the target code syntax. The abstract syntax of the executable code follows:

$$\begin{aligned}
\mathit{Statement} ::= \;& \textbf{if}\ \mathit{Expr}\ \textbf{then}\ \mathit{Statement}\ [\textbf{else}\ \mathit{Statement}] \mid \\
& \mathit{Statement}\ ;\ \ldots\ ;\ \mathit{Statement} \mid \\
& \mathit{Variable} := \mathit{Expr} \mid \\
& \textbf{case}\ \mathit{Variable}\ \textbf{of}\ \mathit{State} : \mathit{Statement} \ldots \mathit{State} : \mathit{Statement} \mid \\
& \textbf{call}\ \mathit{Event} \mid \\
& \textbf{var}\ \mathit{Variable} = \mathit{Expr}\ ;\ \mathit{Statement}
\end{aligned}$$

pState generates code for PIC16F6xx in C or assembly language, and Libelium/Arduino code for ATmega1281 micro-controller. Both are 8-bit RISC-based micro-controllers.

7.6.1 PIC C Code

All executable files can be divided into two groups, (1) generated files and (2) prewritten files. *pState* generates the file *charts.c*, which defines the behavior of the application. Prewritten files *main.c, setupProcessor.c, Scheduler.h, actions.h* can be divided into two groups: target independent, and target dependent files, similar as for the assembly files shown in Fig. 7.9. Target independent files are *main.c*, and *Scheduler.h*. The file *main.c* defines the entry of the application, and initializes variables, chart states, and the scheduler. Then it enters an infinite loop which

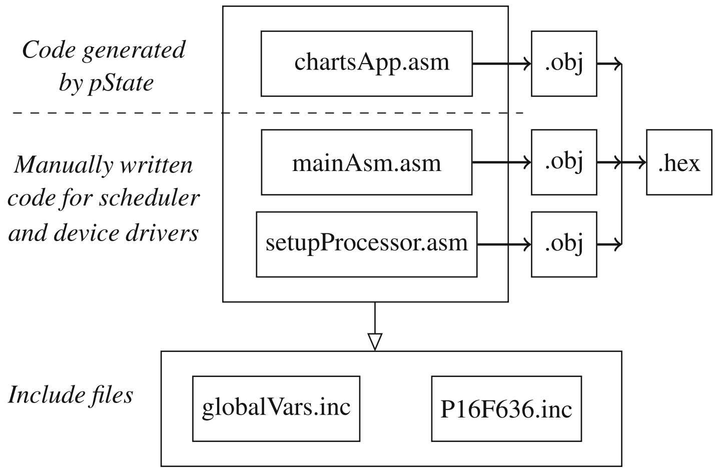


Fig. 7.9 Structure of target code

processes input events and schedules actions. The file *Scheduler.h* defines a data structure which holds timed events and defines functions to schedule and cancel timed events.

The target dependent file *setupProcessor.c* contains routines for processor input/output initialization, timer initialization, etc. The file *actions.c* specifies how external events will be processed. For instance, if a digital signal is connected to PORTA bit 0 of the PIC16F6xx micro-controller, and if the presence of a signal means high voltage on the pin, then that should be defined in *actions.c* as *#define SIGNAL* ($RA0 == 1$).

7.6.2 PIC Assembly Code

Assembly code is created by translating the abstract executable code. Translation of **if-then-else** and **case** statements is straightforward. Most micro-controllers have an instruction which allow constants to be added immediately. In this approach generation of the code is delayed until the *mode* of an expression is known, which is known as *delayed code generation* [25].

We are using CBLOCK 0x20 or CBLOCK 0x40 to allow the variables declared within the block to automatically increment to the next general register, starting from 0x20 or 0x40. Address 0x40 and beyond are used for constants associated with state names, while 0x20 to 0x39 are used for variables.

Names of variables in assembly code are generated as lowercase letter state names. Variables are either integer subranges or Boolean. The generated code consists of state and variable declarations, assignments and expressions, state transitions, macros, statements, and timed transitions. Scheduler, initialization, and I/O actions are not generated from the specification, they are *write-once* code. In this way we have full control over the structure of the application, similar to the approach described in [26].

Code generation depends not only on individual symbols but also on the values of their attributes. We use a one-pass generation that delays emitting the code until the attributes are known [25]. The generated code depends on the fact if the value is held in a register or it is a known constant. If it is constant, the generated code will be smaller since the value does not need to be stored to working register before the operation is performed. Where the value is stored and how it is to be accessed is indicated by attributes of expressions. In our implementation we have the following attribute modes: *Reg* - special function registers, i.e. PORTA, STATUS, etc., *Var* - general purpose registers, *Const* - constant, *AccW* - working register or accumulator. In addition to those we have special modes for expressions like *VarPlusConst*, *VarMinusConst* which indicate operations of addition or subtraction between factors of type variable and constant. Once we know mode of an expression, optimized code can be generated.

7.6.3 Energia, Arduino-Like Code

Arduino is a C-derived programming language. Energia is an Arduino-like IDE for TI LaunchPad (Tiva C) development board. In our implementation, the target is the TM4C123 ARM micro-controller. The program is structured as two routines, *setup*() and *loop*(). The *setup*() routine contains the initialization of variables and is run only once. The *loop*() routine is then executed continuously, allowing variables to change and the program to respond to and control the board. The code can be compiled on the Energia IDE. Custom routines in Energia can be written to perform reoccurring tasks. They are declared like functions in C/C++, with function return type, name, and parameters. In our implementation we assume that no value is to be returned, so the event function type is *void*.

The code generated from the example in Fig. 7.10 is in Fig. 7.11. All states of the hierarchical structure are nested in the root state, which is declared as the variable *root*. pCharts allows direct declaration only of integer subranges and Boolean variables.

Functions that are unique to the Energia language and used to configure, read, and write specific ports of the micro-controller can be called in the body of transitions. Those functions have to be prewritten. They are ignored for the purpose of verification. If we need to set up some pin to be INPUT or OUTPUT, that is done by the *pinMode(pin,mode)* function; to read digital pin value, which can be HIGH or LOW, the function *digitalRead(pin)* is used, and to write to pin *digital-Write(pin,value)* is used. Handling an analog pin is done by *analogRead(pin,value)* and *analogWrite(pin,value)*. It reads and write the value from a specified analog pin with 10-bit resolution.

Untimed event can be executed by (1) polling the trigger of the event or (2) assigning external interrupt to the event. Polling can be done in a continuous loop or by a timer. In our implementation we call the *dispatcher* function in the loop to check if external trigger that causes the *On* or *Off* event is present. The same functionality can be achieved by calling *dispatcher* after a predefined amount of time (i.e. every 1ms). In the prewritten code of Fig. 7.12, the function that configures hardware, HW_Init(), and the function *dispatcher* are shown.

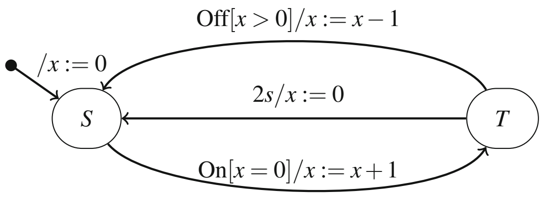


Fig. 7.10 Simple switch operation

```
/*
* Energia (Arduino) code generated from pCharts
*/

#include "OneMsTaskTimer.h"

/* Variables */
#define T 0
#define S 1

int x;
int root;

OneMsTaskTimer_t teExactly0={2000, exactly0, 0, 0};

void exactly0 (){
    if ((root==T)) {
        x=0;
        root=S;
    }
}
void Off(){
    if ((root==T)&&(x>0)) {
        x=(x-1);
        root=S;
        OneMsTaskTimer::remove(&teExactly0);
    }
}
void On(){
    if ((root==S)&&(x==0)) {
        x=(x+1);
        root=T;
        OneMsTaskTimer::add(&teExactly0);
    }
}

void setup(){
    /* Initialization */
    HW_Init();
    root=S;
    x=0;
    OneMsTaskTimer::start(); // Start timer
}

void loop(){
    dispatcher();
}
```

Fig. 7.11 Generated code for the chart in Fig. 7.10

```
void HW_Init(){
  // Initialize the pushbutton pin as an input
  pinMode(PUSH1, INPUT_PULLUP);
  pinMode(PUSH2, INPUT_PULLUP);
}

void dispatcher (){
   noInterrupts ();
  if ( digitalRead (PUSH2)==LOW) {
    On();
  }
  if ( digitalRead (PUSH1)==LOW) {
    Off();
  }
   interrupts ();
}
```

Fig. 7.12 Prewritten hardware-related code, target TM4C123 micro-controller

7.7 Contention Resolution in DASH-7 ISO/IEC 18000-7.2

The ISO/IEC 18000-7.2 [27] standard provides an air interface implementation for wireless, non-contact information system equipment for *item management* applications. The RFID equipment is composed of two principal components: tags and interrogators. We study a *system* with *active* tags, i.e. tags with own source of energy, like battery. Each tag has a unique serial number and other data. It is intended for attachment to a managed item. An interrogator is a device that communicates to tags in its RF communication range. The interrogator controls the master–slave protocol, reads information from the tag, directs the tag to store data, ensures message delivery and validity. We present the method by which an interrogator identifies and communicates with one or more tags present in the operating field of the interrogator over a common radio frequency channel. Tags do not transmit unless commanded to do so by the interrogator. An interrogator can communicate with tags individualy, or with the tag population as a whole.

7.7.1 Tag Collection and Collision Arbitration

The tag collection process is an iterative process that includes methods for coordinating responses from the tag population and handling collisions which occur when multiple tags transmit at the same time. The entire tag collection process is referred to as a *complete collection sequence*. Figure 7.13 shows a complete collection sequence consisting of a *wakeup period* (WP) followed by a series of *collection*

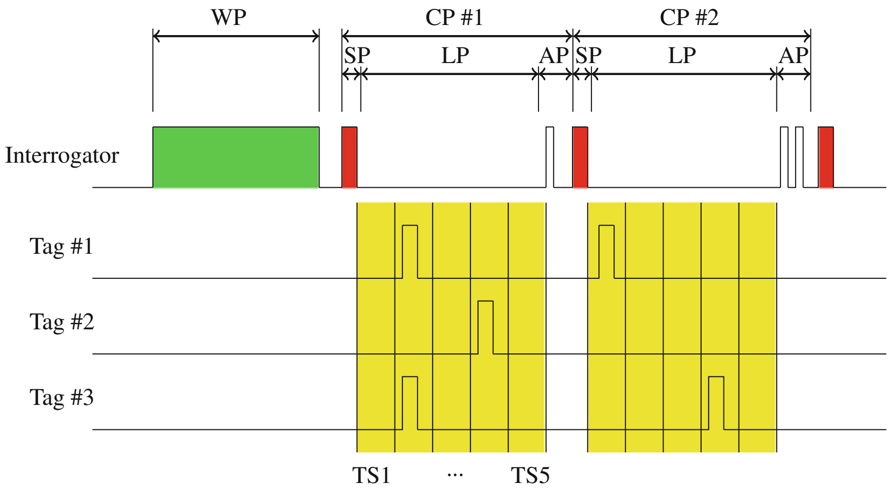


Fig. 7.13 Interrogator-tag communication timing diagram

periods (CP). Each collection period consists of a *synchronization period* (SP), a *listen period* (LP), and an *acknowledge period* (AP). The LP is further divided into multiple time slots (TS).

For three tags and five time slots as shown in Fig. 7.13, in the first communication period, tags #1 and #3 transmit in the same time slot, so there will be a collision. In the first acknowledge period there is an acknowledgment only for the message of tag #2. In the second communication period, tags #1 and #3 retransmit the message, but this time tag #1 transmits in the time slot 1, and tag #3 transmits in time slot 4, so there is no collision, and in the acknowledge period there are two acknowledgement messages.

7.8 Collision Model

We can calculate the collision probability by calculating the number of possible transmissions without collision and divide it by the total number of possible transmissions. For n tags transmitting, the first tag can transmit in any of the m time slots, the second tag should transmit in any of the $m-1$ slots to avoid collision, and so on. The number of transmissions without collision is

$$NC = m \cdot (m-1) \cdot \ldots \cdot (m-n+1) \tag{7.1}$$

while the number of all possible transitions is

$$AT = m \cdot m \cdot \ldots \cdot m = m^n \tag{7.2}$$

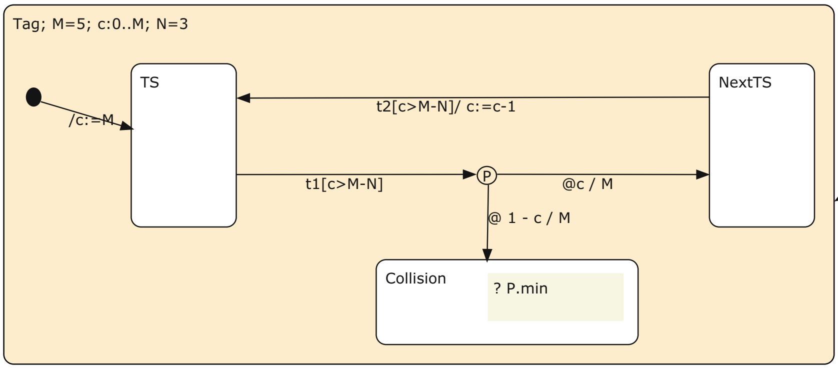


Fig. 7.14 Collision model, three tags, five time slots

The probability that a collision will happen is simply

$$1 - NC/AT \tag{7.3}$$

We assume a model of three tags $N = 3$ and five time slots $M = 5$. The number of possible transmissions without collisions is $NC = 5 \cdot 4 \cdot 3 = 60$, and the number of possible transmissions is $AT = 5 \cdot 5 \cdot 5 = 125$. The probability of at least one collision according to (7.3) is 0.52.

The collision model represented by pCharts is shown in Fig. 7.14. In the *Collision* state, by "? *P.min*″ we query the collision probability, or probability to go to *Collision* state, which is calculated as 0.52. To calculate the collision probability for a different number of time slots or a different number of tags, all we have to do is to assign new numbers to *M* or *N* in *Tag* state declaration.

7.9 Collection Period Power Consumption

The collision model in Fig. 7.14 is without timed transitions and the generated input code for the model checker is an MDP. But, in the power consumption model, we need to know how long tags stay in states and the current consumption in those states. The model of power consumption is shown in Fig. 7.17. From this model, *pState* generates a PTA.

On the PTA model we can query the average power consumption in one collection period (CP), taking into account the collision probability calculated on the collision model. The current consumption for a typical active tag during *transmission* is 9.2 *mA*, in *receiving* mode 0.2 *mA*, and in *standby* 0.0024 *mA* [28]. In the construction of the transitions we use data from Fig. 7.15. In the case of three tags and five time slots, the collision probability is 0.52, or 52 %, so the transition

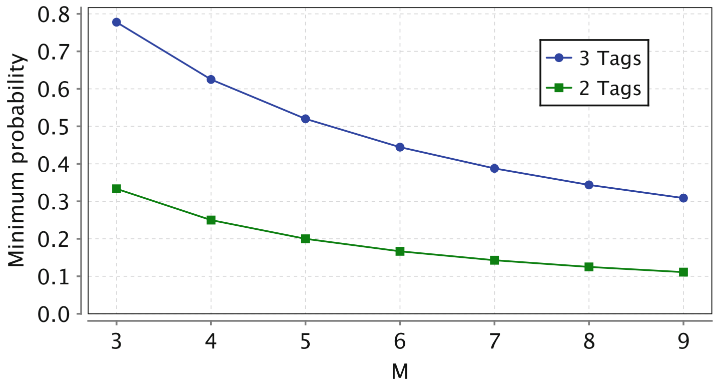


Fig. 7.15 Collision probability for tags $N = [2, 3]$ and time slots $M = [3..9]$

```
mdp

const M = 5;
const N = 3;
const Collision =0; const NextTS=1; const TS=2;

module collision
      tag :[0..2]   init TS;
      c :[0.. M] init M;

      [t2] (tag=NextTS)&(c>(M−N)) −> (c'=(c−1))&(tag'=TS);
      [t1] (tag=TS)&(c>(M−N)) −> (1−(c/M)):(tag'=Collision) + (c/M):(tag'=NextTS);
endmodule
```

Fig. 7.16 PRISM code generated by *pState* for model shown in Fig. 7.14

from state *Tx* to *Next* is probabilistic with 52 % probability. That means if a collision happens, the tag needs another collection period to perform the operation. In the case of collision, the probability that all three tags select the same time slots is 4 %, which is modelled by a probabilistic transition from *Tx* to *ThreeCollisions* state. If there is a collision of two tags, in the next collection period, on five time slots, only those two tags are retransmitting. According to Fig. 7.15, the collision probability is 20 %, and that is represented by probabilistic transition from *TwoTags* to *Next* state. The maximum number of collection periods in the model is three (Fig. 7.16).

The queries are verified over the formulae $R\{``cons''\}max = ?[F(tagconsum = End)]$ and $Pmax = ?[F(tagconsum = End)\&(i = 3)]$. For the first, the calculated value for the electrical charge is 280.72 *mAms* (Fig. 7.17). The probability of not receiving all tags in three collection periods is calculated as 0.1106, or 11.06 %, so about 89 % of time all tags are read in three collection periods (Fig. 7.18).

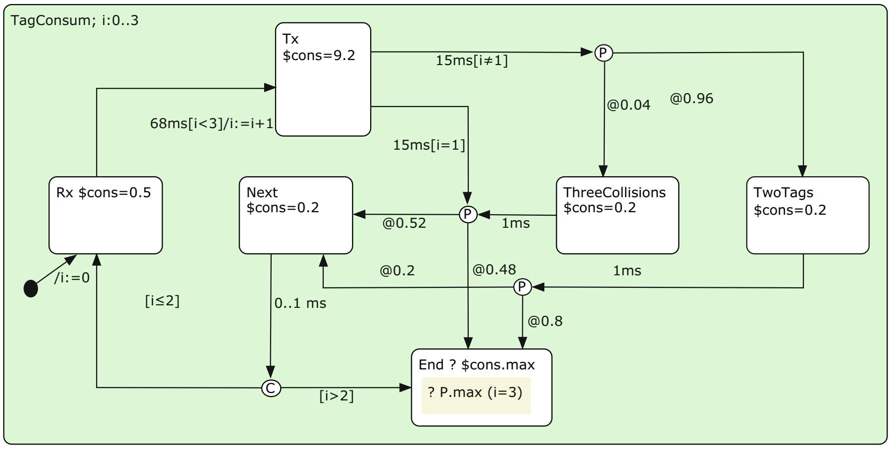


Fig. 7.17 Collection period power consumption

7.10 Executable Tag Code

Figure 7.19 gives the tag operation model. It has two parallel processes, *Tag*, which represent tag operation, and *Mode* which represents tag transition from sleeping to working mode and back. Initially, *Mode* is in *Sleep*, in which periodically, every 1*ms*, the presence of a wakeup signal is checked. If there is no signal, it goes back to *Sleep* and repeats the process after 1*ms*. If there is a wakeup signal, the indicator *field* is set to *true*, and the system goes first into *Field* and immediately to *FieldON*. From that state it goes to *Work* and broadcasts the event *WakeUp*. That event moves *Tag* from *Start* into *Preamble*. On that transition, procedure *WAKEUP* is called. It has to recognize the (*WP*) preamble, Fig. 7.13, and has to be executed in 2.45 to 4.8 seconds. Next, *Tag* goes into *CP* state in which it receives a command form the interrogator, and goes into listen period *LP*, in which the tag transmits its message in a randomly selected time slot. In the *Ack* state, *Tag* waits for confirmation or acknowledgment message. If the received command informs the tag that communication is *done*, it goes to *Finished* and then back to *Start*. On the transition form *Finished* to *Start*, local event *GoToSleep* is broadcasted. This forces the parallel task *Mode* to go from *Work* to *Sleep* mode. Another way to go from *Work* to *Sleep* is on timeout, which is 30 seconds according to the protocol specification.

From the specification we generate the framework for tag application according to the DASH-7 protocol specification. For the full implementation it is necessary to implement the following subroutines (1) *WAKEUP* to detect the low frequency wake up signal, (2) *RECEIVECP* to receive broadcast and point-to-point commands, (3) *LISTENPERIOD* to send a packet message which contains a unique tag identification number to the interrogator in the selected time slot, (4) *ACKPERIOD* to receive an acknowledgement from an interrogator. Those routines should satisfy timing requirements from the general framework model. Each transition can

```
pta

const Next=0; const Rx=1; const Tx=2; const TwoTags=3; const End=4; const
        ThreeCollisions =5;

module powerconscp2
      tagconsum :[0..5]  init  Rx; tagconsumclk : clock;
      i :[0..3]  init  0;

      invariant
            (tagconsum=Next=>tagconsumclk<=1)
            & (tagconsum=Rx=>tagconsumclk<=68)
            & (tagconsum=Tx=>tagconsumclk<=15)
            & (tagconsum=TwoTags=>tagconsumclk<=1)
            & (tagconsum=ThreeCollisions=>tagconsumclk<=1)
      endinvariant

      [] (tagconsum=ThreeCollisions)&(tagconsumclk=1) −> 0.52:(tagconsum'=Next)
            &(tagconsumclk'=0) + 0.48:( tagconsum'=End)&(tagconsumclk'=0);
      [] (tagconsum=Tx)&(i=0)&(tagconsumclk=15) −> 0.52:(tagconsum'=Next)
            &(tagconsumclk'=0) + 0.48:( tagconsum'=End)&(tagconsumclk'=0);
      [] (tagconsum=Next)&(tagconsumclk>=0)&(tagconsumclk<=1) −>
            (tagconsum'=(i<=2)?Rx:End)&(tagconsumclk'=0);
      [] (tagconsum=Tx)&(i!=0)&(tagconsumclk=15) −> 0.96:(tagconsum'=TwoTags)
            &(tagconsumclk'=0) + 0.04:( tagconsum'=ThreeCollisions)&(tagconsumclk'=0);
      [] (tagconsum=Rx)&(i<3)&(tagconsumclk=68) −> (i'=(i+1))&(tagconsum'=Tx)
            &(tagconsumclk'=0);
      [] (tagconsum=TwoTags)&(tagconsumclk=1) −> 0.8:(tagconsum'=End)
            &(tagconsumclk'=0) + 0.2:( tagconsum'=Next)&(tagconsumclk'=0);
endmodule

rewards "cons"
      (tagconsum=Next): 0.2;
      (tagconsum=Rx): 0.5;
      (tagconsum=Tx): 9.2;
      (tagconsum=TwoTags): 0.2;
      (tagconsum=ThreeCollisions):  0.2;
endrewards
```

Fig. 7.18 PRISM code generated by *pState* for model shown in Fig. 7.17

be automatically transferred into assembly code and worst case execution time (WCET) can be calculated in terms of processor cycles. Assembly code generation and calculation of WCET on pCharts is described in [29].

From pCharts in Figs. 7.14 and 7.17, input code for the probabilistic model checker is generated, and from the pCharts in Fig. 7.19, executable code is generated. All generated code is posted on the *pState* web site.

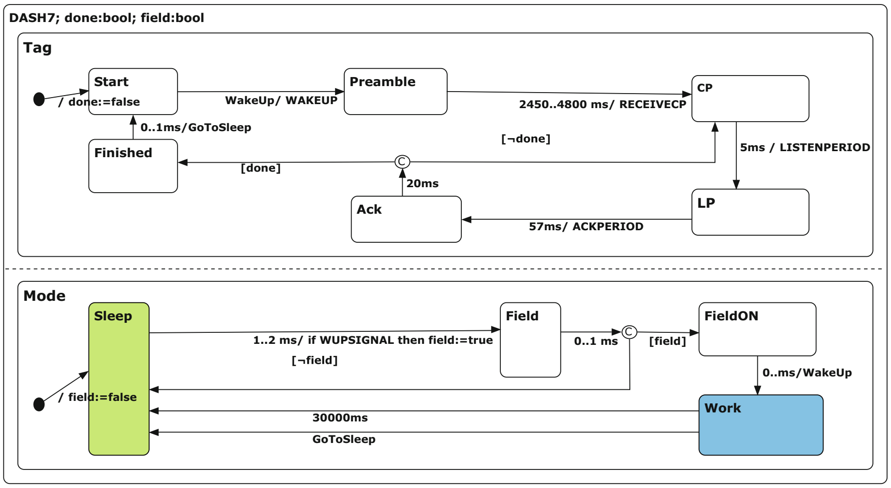


Fig. 7.19 Tag model

7.11 Conclusion

In this chapter, we presented the model based process of system analysis and code generation. From the same model, input code for probabilistic model checker and an executable code are generated. Probabilistic model checker is used for quantitative and qualitative properties analysis. The target for executable code are 8- and 16-bit micro-controllers used in embedded systems. The code is generated in C or assembly language. As a part of assembly executable code generation, we can calculate execution time calculation for each graph transition. Target micro-controllers do not have features like multi-stage pipelines and caches, so execution time in a number of executable *cycles* is actual execution time. The integrated process of model-based analysis and code generation increases an accuracy of analysis and fidelity of generated executable code.

We created a tool, *pState*, for the purpose of *holistic* software design. In addition to executable code generation, the tool is used to verify quantitative properties. This is of practical interest specially for complex embedded systems where not only functional correctness and timing guarantees are relevant, but also quantitative properties, which cannot be analyzed by considering exclusively the software part. The environment has to be considered as well.

In the example of RFID tag working according to DASH-7 ISO/IEC protocol, we show how one tool can be used for system property analysis (collision probability), device property analysis (power consumption), and device code generation.

The goal is an *automated* approach from modelling and analysis to code generation. This can be used to *evaluate* design alternatives and generate *trustworthy* code.

References

1. A. Hinton, M. Kwiatkowska, G. Norman, D. Parker, PRISM: A tool for automatic verification of probabilistic systems. In *Proc. 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, vol. 3920 of *LNCS*, ed. by H. Hermanns, J. Palsberg (Springer, New York, 2006), pp. 441–444
2. D. Harel, Statecharts: A visual formalism for complex systems. Sci. Comput. Program. **8**(3), 231–274 (1987)
3. D. Harel, Statecharts in the making: a personal account. Commun. ACM **52**(3), 67–75 (2009)
4. B. Nokovic, E. Sekerinski, pState: A probabilistic statecharts translator. In *Embedded Computing (MECO), 2013 2nd Mediterranean Conference on*, pp. 29–32, 2013
5. B. Nokovic, E. Sekerinski, Verification and code generation for timed transitions in pCharts. In *Proceedings of the 2014 International C* Conference on Computer Science #38*, C3S2E '14 (ACM, New York, NY, USA, 2014), pp. 3:1–3:10
6. D.N. Jansen, *Extensions of Statecharts with Probability, Time, and Stochastic Timing*. PhD thesis, University of Twente, Enschede, 2003
7. E. Sekerinski, Design verification with state invariants. In *UML 2 Semantics and Applications*, ed. by K. Lano (Wiley, New York, 2009), pp. 317–347.
8. W. Randelshofer, JHotDraw. http://www.randelshofer.ch/oop/jhotdraw/index.html, December 2012
9. R. Eshuis, D.N. Jansen, R. Wieringa, Requirements-level semantics and model checking of object-oriented statecharts. Requir. Eng. **7**(6), 243–263 (2002)
10. D. Harel, A. Naamad, The statemate semantics of statecharts. ACM Trans. Softw. Eng. Methodol. **5**, 293–333 (1996)
11. E. Sekerinski, R. Zurob, iState: A statechart translator. In *UML 2001 - The Unified Modeling Language, 4th International Conference, Lecture Notes in Computer Science 2185*, ed. by M. Gogolla, C. Kobryn, Toronto, Canada, 2001, pp. 376–390
12. E. Mikk, Y. Lakhnech, M. Siegel, G.J. Holzmann, Implementing statecharts in PROMELA/SPIN. In *Proceedings of the Second IEEE Workshop on Industrial Strength Formal Specification Techniques*, WIFT '98 (IEEE Computer Society, Washington, DC, USA, 1998), pp. 90–101
13. E. Sekerinski, Verifying statecharts with state invariants. In *Proceedings of the 13th IEEE International Conference on Engineering of Complex Computer Systems* (IEEE Computer Society, Washington, DC, USA, 2008), pp. 7–14
14. E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley Longman Publishing, Boston, MA, 1995)
15. University of Birmingham, Probabilistic symbolic model checker. Website, 2015. http://www.prismmodelchecker.org/
16. M. Fruth, *Formal Methods for the Analysis of Wireless Network Protocols*. PhD thesis, University of Oxford, 2011
17. C. Baier, J.P. Katoen, *Principles of Model Checking* (MIT Press, New York, 2008)
18. G. Norman, D. Parker, J. Sproston, Model checking for probabilistic timed automata. Formal Methods Syst. Des. **43**(2), 164–190 (2013)
19. R. Alur, D.L. Dill, A theory of timed automata. Theor. Comput. Sci. **126**, 183–235 (1994)
20. C.A.R. Hoare, Communicating sequential processes. Commun. ACM **21**, 666–677 (1978)
21. V. Forejt, M. Kwiatkowska, G. Norman, D. Parker, Automated verification techniques for probabilistic systems. In *Formal Methods for Eternal Networked Software Systems (SFM'11)*, vol. 6659 of *LNCS*, ed. by M. Bernardo, V. Issarny (Springer, New York, 2011), pp. 53–113
22. M. Kwiatkowska, G. Norman, D. Parker, J. Sproston, Performance analysis of probabilistic timed automata using digital clocks. In *Formal Modeling and Analysis of Timed Systems*, vol. 2791 of *Lecture Notes in Computer Science*, ed. by K.G. Larsen, P. Niebert (Springer, Berlin, Heidelberg, 2004), pp. 105–120

23. M. Kwiatkowska, G. Norman, D. Parker, Stochastic games for verification of probabilistic timed automata. In *Proceedings of the 7th International Conference on Formal Modeling and Analysis of Timed Systems*, FORMATS '09, ed. by J. Ouaknine, F.W. Vaandrager (Springer, Berlin, Heidelberg, 2009), pp. 212–227
24. A. Bianco, L. de Alfaro, Model checking of probabalistic and nondeterministic systems. In *Proceedings of the 15th Conference on Foundations of Software Technology and Theoretical Computer Science*, ed. by P.S. Thiagarajan (Springer, London, UK, 1995), pp. 499–513
25. N. Wirth, *Compiler construction*. International computer science series (Addison-Wesley, Reading, 1996)
26. IARSystems, IAR visualstate concept guide, 1999
27. DASH7 Alliance, Dash7, 2013. http://www.dash7.org/
28. M. Paun, Posttag PT23 technical specification. Technical report, Lyngsoe Systems, 2006
29. B. Nokovic, E. Sekerinski, Model-based WCET analysis with invariants. In *Proceedings of the 15th International Workshop on Automated Verification of Critical Systems (AVoCS 2015), Edinburgh, UK, Sep., 2015*, vol. 72 of *Electronic Communications of the EASST*, ed. by G. Grov, A. Ireland, 2015

Chapter 8
Positioning System for Recreated Reality Applications Based on High-Performance Video-Processing

Patricia Martinez and Eugenio Villar

8.1 Introduction

While Moore's Law is still in place, the complexity of embedded systems continues to grow exponentially. Embedded Systems are implemented on complex HW/SW platforms, requiring more powerful design methods and tools. Raising the level of abstraction in which the system is modeled has been proposed to allow the analysis and optimization of the system at earlier stages of the design process. Implementation of complex video processing algorithms on a high-performance, multi-core, heterogeneous platform constitutes the kind of system design where new design methods and tools are necessary.

Model-driven development (MDD) [15] has been proposed to capture the initial, high-level model of the system [4]. The adoption of standard languages and profiles, like Unified Modeling Language (UML) [18] and the Modeling and Analysis of Real-Time Embedded Systems (MARTE) [11–13], contributes to a standard graphical representation and the reusability and interoperability of the models. In this chapter, MDD based on UML/MARTE is used in the modeling of a positioning system for "recreated reality" applications.

Although the term "virtual reality" applied to the theater experience is more than 75 years old, its meaning has evolved dramatically since that time with the evolution of technologies able to support virtual reality (VR) experiences. The evolution of computers made possible the current meaning of virtual reality as the use of computing technology to create simulated 3-dimensional (3D) worlds in which the user can move and interact while creating the experience of being inside these worlds. The 3D world is created and displayed depending on the position

P. Martinez • E. Villar (✉)
TEISA Department, University of Cantabria, ETSI. Industriales & Telecom., Avda. Los Castros s/n, 39005 Santander, Spain
e-mail: villar@teisa.unican.es

A. Molnos, C. Fabre (eds.), *Model-Implementation Fidelity in Cyber Physical System Design*, DOI 10.1007/978-3-319-47307-9_8

of the user using specialized graphic SW tools [2]. The 3D experience is only partial in traditional personal computers and video-consoles where the user sees the 3D environment through the screen of the TV or the computer monitor. The term “virtual reality” is usually associated with “immersion,” that is, the sensation of being actually inside the virtual world with the world changing as the user moves the head and looks in any direction around. The two main ways to create the immersive experience are the computer assisted virtual environment (CAVE) and VR glasses.

A CAVE is a room in which the virtual environment is created by projecting it onto the walls of the room. The projection also covers the floor and the ceiling improving the immersive experience. By using similar glasses to those used in 3D cinemas, the CAVE can create 3D environments. In any case, the cost associated with the required infrastructure limits the applicability of this technology to a reduced number of specific domains such as product development and prototyping and collaborative planning in sectors such as construction.

Improvements in video display technology during the last years, especially liquid-crystal displays (LCDs), have made 3D virtual glasses available at prices low enough to reach a large number of customers. The number of applications has increased dramatically during this time. The first and still the most important is video gaming. With VR glasses the user can fully obtain an immersive sensation of being part of the game he/she is playing. To do so, the glasses come with a gyroscope able to detect the orientation of the head and therefore adapt the 3D graphic environment point of view accordingly. A second, very promising application field is education. With VR glasses, the whole class can move to any virtual place the teacher selects in order to show in a realistic, immersive way the subject matter to learn, from real or reconstructed buildings of any age anywhere, geographical landscapes and wildlife under different climates and in different times, etc. Some glasses make use of the screen of a mobile phone instead of internal displays. To do so, the screen of the phone is divided into two parts, left and right, each one to be seen by an eye thus creating 3D images. As this avoids any additional electronics, the glasses can be inexpensive [3]. An additional advantage is that the glasses become autonomous avoiding the need to be connected to a PC or tablet.

In some applications, the glasses just reproduce the video captured by cameras elsewhere, leading to many different possibilities. One is being part of something occurring in a different location, even miles away [14]. If the cameras are directly put in front of the glasses, then, one may live someone else’s life [1].

Another technology which has gained interest in last years is “augmented reality” (AR). In AR applications, the real image is always seen but reality is enriched with additional information, usually meta-information. The glasses used in this case are different to those for VR. In order to “add” information to the living scene, small projectors show the computer-generated information on the surface of the glasses. In this way, the user sees both the real image and the “added” one [17]. This technology may lead to a completely new human–computer interaction [7].

8.1.1 Recreated Reality

The advantage of 3D glasses equipped with cameras is that they give complete control of the synthetic image provided to the user generated as a combination of both reality and virtual reality. We will call this new, different possibility, "recreated reality" (RR). The main difference against AR is that now, the complete image can be synthetic. In order to take most advantage from this technology it is important to allow the user to move freely inside the RR space. The virtual reality provided to the user should correspond to the movement of the user inside the real space. While the gyroscope in the glasses will track the watching direction and, therefore, which part of the virtual reality scene to show from the position of the user, the scene itself should change as the user moves. As a consequence, correct positioning of the user inside the RR space is an essential component of the RR technology.

The positioning technology should be as flexible as possible. In some cases the space may be small. Other applications may require large spaces. In some cases the space is indoors, but other cases may require an outdoor space. In some cases, the space may be illuminated by artificial light while in others the application may require sunlight. A positioning system able to work in such varied scenarios is required.

The general architecture of the proposed RR system is shown in Fig. 8.1. The positioning system proposed calculates the position from the analysis of the images captured by the cameras in the glasses. This information, along with the direction of the glasses, is sent to the graphic software which will generate the image to be seen at this position looking in that direction. This information may be added to the real image captured by the cameras. The resulting RR is shown to the user through the 3D glasses.

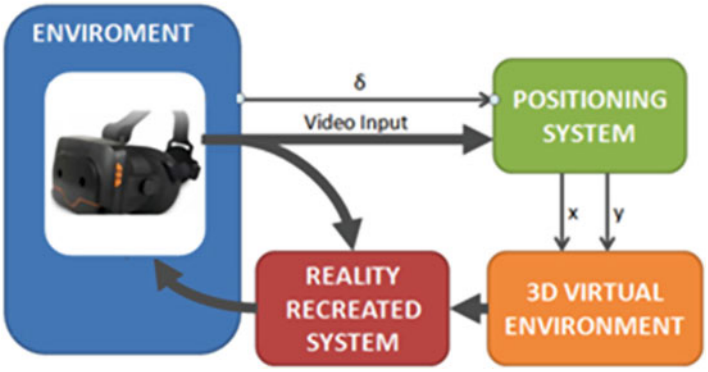


Fig. 8.1 Architecture of the RR system

8.1.2 State of the Art in Positioning Systems

In the last years, there has been a growing interest in systems and products related to the location of objects in three dimensions (3D). The main reason for this derives from the large number of sectors where this technology can be applied, such as robotics, training, medicine and gaming, among many others.

To establish the precise position of the individual, the systems may be based on the use of many different means such as cameras, optical sensors, accelerometers, gyroscopes, GPS, etc. In the event that vision systems are used, it is necessary to perform a pre-processing of the region of interest where the individual is located using imaging algorithms that detect corners, edges, or reference markers; then, with these data it is possible to obtain the real 3D coordinates of the environment.

When only a reference marker is visualized, the use of stereo cameras and epipolar geometry is necessary. The characterization of a point in a three-dimensional space requires knowledge of its coordinates (x, y, z) within the environment where it is situated, relative to a reference position. The most common technique is based on the use of two or more calibrated cameras that provide left and right images of the same scene. The 3D coordinates of the point or object can be estimated using stereo mapping, that is, looking for the same point in both images and then applying projective or epipolar geometry (describing the relationship between the image planes of the cameras and the point).

When there is more than one marker or pattern available, it is possible to apply other techniques to locate the object on the stage. Through triangulation, knowing the actual distance between markers, it is possible with only two markers and a single camera to obtain the parameters to establish the target position in the environment. This practice simplifies the computational cost, because it is not necessary to analyze two images and their matches, but it requires greater precision in detecting markers.

One of the problems that has been identified [9] when using markers is the illumination that is required in the environment where the images will be taken. The points to be detected in the scene may be lost because of darkness. This limits the use of this system to daylight outdoors or indoors under controlled brightness. Furthermore, this technique requires the use of contrast enhancement algorithms and a database where the specific markers are stored, which substantially increases the computing time of these systems. In addition, it significantly limits the distance between printed markers and the user, unless its size is large enough for capture by the image sensor. To solve this problem, light markers working in the visible or infrared spectrum can be used, such as light emitting diodes or lasers. In some cases [5] the use of light sources with varying luminance or pulsed light has been proposed, which can cause synchronization failures. Still, the use of light markers can pose problems, particularly in environments where light sources with much higher luminance than the marker itself (in the worst case, sunlight) or where sources emit radiation in the same direction. In these situations, the image sensor is not able to differentiate between one light source and another, so it will oblige, like in the previously case, this technology to be used in bright environments without bright light sources in them. Therefore, it is necessary to improve the positioning systems to prevent the light conditions in the environment having such a great effect.

In [8], the authors propose incorporating infrared luminous markers on a tape on the user's head. To do this, they put two independent cameras in the scenario, which require a synchronization process to make the shot simultaneously, located at a distance equal to the length of the wall of the room where they are tested. The algorithm used to make an estimation of the position is based on stereo correspondence. One of the drawbacks is that it cannot be used for augmented reality systems or simulated, because the cameras do not show what the user sees, besides being restricted to indoor environments with limited dimensions.

Other studies [6] propose the use of infrared markers located on the wall of a room to locate the user. Specifically, they consider the use of two types of markers: active and passive. The active markers are formed by a set of three infrared LEDs and a signal transmitter that sends data from its actual position to a decoder that the user carries. The passive markers have only one source of infrared light, from which they obtain the relative user position. In addition to receiving signals from active markers, the system calculates the relative distance between the user and the markers using stereo vision. This technique, as occurred in the cases discussed above, is restricted to indoor use.

There are other methods that do not require direct vision of one or more cameras with reference markers for locating and tracking individuals. RF techniques involve measuring distances from static or moving objects by emitting electromagnetic pulses that are reflected from a receptor. These electromagnetic waves are reflected when there is significant difference in atomic density between the environment and the object, so the technique works particularly well in cases of conductive materials (metals). They are able to detect objects at greater distances than other systems based on light or sound; however, they are quite sensitive to interference or noise. It is also difficult to measure objects located at different distances from the transmitter, because the pulse frequency will vary (lower the farther away and vice versa). However, there are experimental studies which are able to demonstrate the use of this technique to estimate the user location with a high level of accuracy.

Another example of existing solutions is the LIDAR system, which calculates the distance through the time taken for a light pulse to be reflected on an object or surface; using a device with a pulsed laser as a light emitter and a photo-detector as a receptor of the reflected light. The advantages of these systems are the accuracy achieved over large distances (using lasers with wavelength > 1000 nm) and the possibility of mapping large areas by scanning light pulses. The disadvantages are that it is necessary to analyze and process each point, and the difficulty of automatically reconstructing three-dimensional images.

A similar method is used by the HTC and Valve Corporation Vive device, a virtual reality head-mounted display. The technology requires two lighthouse stations (a base station with two spinning IR lasers and a bank of infrared LEDs) in an indoor environment. The stations will function as emitters; the receptor is composed of a headset and two wireless controllers with 70 sensors (gyroscopes, accelerometers, VR visor, etc.). The system computes the user's position with respect to the reference point and the lighthouse station transmits a flash from the LEDs and a pulsed light from the laser. When the receptor, either on the headset or the wireless controllers, receives the flash, it begins counting the time until it receives the light from the laser. The user position is found from the relationship between the position of the sensor which has detected the light from the laser and the time between the led's flash and the laser's light. Although very effective in reduced areas, this system is very difficult to extend to larger spaces. It is valid in indoor environments with a maximum area of 25 m^2. The lighthouse stations must be situated in a high position to avoid occlusion and to cover the entire scene. Therefore, the scene should be free of objects. Its extension to non-uniform spaces, like a house, would not be without problems.

Another method of tracking a target using a laser-based technique is that used by Microsoft's Kinect. This device is a motion sensing apparatus, consisting of an RGB camera and a depth sensor. The sensor features an infrared laser projector used as an emitter and a monochrome CMOS sensor or 3D depth camera used as a receptor. It obtains the depth map of the scene under any ambient light conditions. Many applications are based on this device. They use a combination of RFID identification and positioning technology, Kinect's vision for the position of a person [20] and a single RFID reader [10] situate a Kinect in a small environment or several Kinects to cover all the area. However, this sensor has distance limitations, so it needs to use more than one device to cover a scene, at least four apparatus for 10 m^2, which would imply expensive systems for big environments.

The objective technical problem that arises is therefore to provide a system for detecting the position and orientation of a person or object in any environment, large or small, inside or outside, whatever the lighting conditions are.

8.2 System Architecture

The UC has patented [19] a system and a method to obtain the spatial location of objects or individuals in a scene under all environment conditions (indoors or outdoors) and with greater distances between the user and the marker than other systems. As shown in Fig. 8.2, it is based on the use of the following main components:

1. Light sources used as markers to calculate relative positions of the industrial machinery/ robot;
2. A stereo camera to display these markers in the image of the scene;
3. An angle measuring device (such as a gyroscope or electronic compass) to provide angles of rotation of the target object at each instant of time;
4. A digital signal processor, with uses the stored coordinates (from the memory) and the output parameters obtained from the stereo camera and angle measuring device to determine the target object position in the 3D environment.

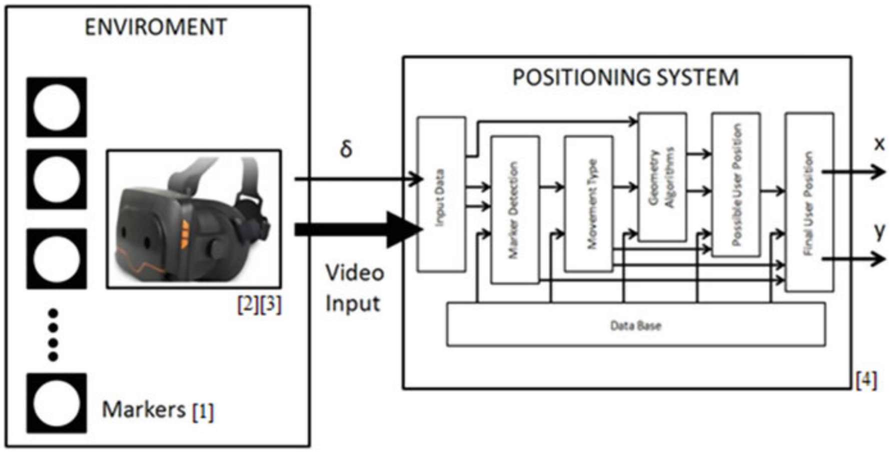


Fig. 8.2 Schematic of the positioning system

The positioning system follows the COMPLEX UML/MARTE modeling methodology [4]. Thus, it is independent of any hardware architecture leading to a platform independent model (PIM), which can be implemented on HW or SW resources.

8.2.1 Reference Marker Description

The system is based on the identification of fiducial markers for location and orientation of moving objects in different kinds of environments. Taking into account the above, it is necessary to use a specific marker which is visible for the camera outdoor and indoor, under bright and dark light conditions and at short and long distances.

Figure 8.3 shows what the proposed reference marker is like. It consists of two main elements, a light source and a black contrast surface. The preferred light source is an LED emitting in the visible range (400–700 nm); this type of source is a point light source that reaches distances greater than 50 m. An LED may also be considered as a non-hazardous material, due to its working optical power and low reaction time of aversion ($\cong$ 250 ms). Nevertheless, the system works properly with other kinds of light sources, as the light receptors and the cameras are usually able to detect these light sources in the visible and infrared spectrums.

Moreover, the contrasting surface must be black, to absorb 100 % of the light; the dimensions of the contrasting surface depend on the lighting conditions of the environment, the luminous flux light source and the maximum distance between the camera and luminous markers. The template or contrast surface lies on the outside

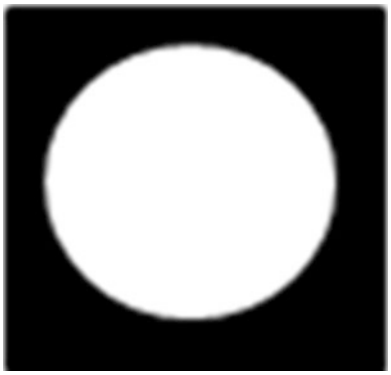


Fig. 8.3 Reference marker

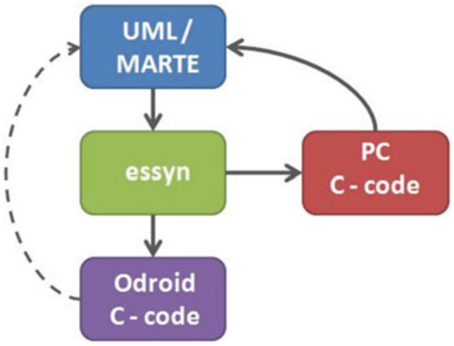
of the light source, specifically on the back, leaving the light source in the camera's view. In the event that the ambient or background light behind the light source is dark enough, it is not necessary to add the contrast surface.

8.2.2 Design Methodology and Workflow

In order to ensure modeling fidelity of the implementation of complex embedded systems on multi-processing platforms, design methodologies that, based on separation of concerns, enable the design teams to work in an efficient way, is required. Separation of concerns enables the specialization of the design process; separate but collaborating sets of designers can deal with different system concerns (application modeling, HW/SW platform design, etc.), improving the development process. Therefore, well-defined system concerns in the same model enable designers to focus on their specific designing domain, guaranteeing system consistency by using the same specification language, producing synergy among different domains.

In order to support all the different stages of the flow, a high-level system modeling and design methodology has been applied [4]. It follows a model-driven architecture (MDA), component-based, software-centric approach. The design-space exploration and implementation activities are developed around the model. Another remarkable feature is that the COMPLEX methodology supports the separation of concerns paradigm, keeping the functional and non-functional aspects well-differentiated. The unified modeling language (UML) is used to capture all the required information of the system functionality, the HW/SW platform and the selected architectural mapping in a single-source. In order to provide UML with the required modeling features, the MARTE profile is used to capture all the specific characteristics related to the embedded system being designed. This methodology can completely describe the system, enabling automatic generation of the input code. More concretely, the complete UML/MARTE model is based on graphical descriptions, which are called views, each one focused on a relevant aspect of the system. These views describe the system functionality, the target platform, and the resource allocation.

Fig. 8.4 Design workflow



The design methodology follows in this project is shown in Fig. 8.4. The first step is to develop the UML/MARTE model. This model is composed of three main parts. Each one, associated with several views ensuring the separation of concerns mentioned above:

- The platform independent model (PIM), which describes the functional and non-functional aspects of the system components independently of where and how these components will be implemented and deployed. Therefore, it can be reused when the application is going to be implemented on other platforms. The following are the associated views:
  - Data View
  - Functional View
  - Application View
  - Communication View
  - Memory Space View
- The platform description model (PDM), which describes the hardware and software resources where the functionality can be mapped. A PDM. The following are the associated views:
  - HW Platform View
  - SW Platform View
- The platform specific model (PSM), which describes the system architecture as the allocation of functional components to platform resources, leading to a concrete system implementation. The following is the associated view:
  - Architectural View

Figure 8.5 shows how all the presented views of the three sub-models are related among them.

Following with the design workflow of Fig. 8.4, once the model is finished we can create easily an executable model of the system in the PC using eSSYN, the SW synthesis tool developed by the University of Cantabria in the Pharaon FP7

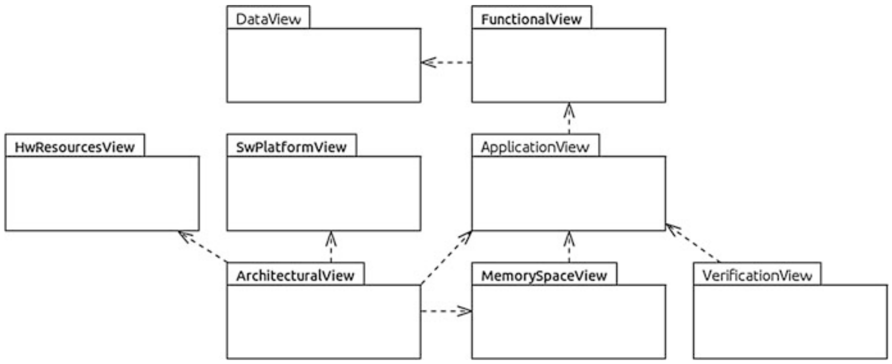


Fig. 8.5 Relations among UML/MARTE views

project [13]. Only when the PC executed in the PC workstation is correct, then it is synthesized using eSSYN again on the Odroid in order to verify its correctness in the final execution platform.

The positioning system, presented in previous section and shown in Fig. 8.2, is divided into six components. In order to describe the communication between components it is necessary to specify the set of services, which are defined in the *Functional View*. These services are grouped into interfaces, which are specific for each inter-component communication channel. The interface services are modeled making use of the functions specified in the DataView. Additionally, this view includes the specification of the files that contain the implementation, that is, the functional source code of each. The functional view of the positioning system is shown in Fig. 8.6, where each interface is represented with the same color as the corresponding component. These components are shown in Fig. 8.7.

The functional components are the basic building blocks of the system application, and are defined in the Application View. As shown in Fig. 8.7, the Application View includes the description of the application components, the relationship among them, and their interconnection through ports by the set of required/provided services defined by the corresponding interfaces.

Each application component defined in this view is associated with the files with the source code (c-code and their heathers) describing its functionality. These are defined in the Functional View (Fig. 8.8).

The platform description model (PDM) describes the HW/SW platform architecture. Later (PSM section) the memory partitions defined in the PIM will be allocated to the resources defined in the PDM.

In our case, the HW Platform is the ODROID-XU3 development board, which has 4 BIG cores (Cortex-A15) and 4 LITTLE cores (Cortex-A7). The SW Platform defines the operating systems available in the HW/SW platform, in our case the Linux OS.

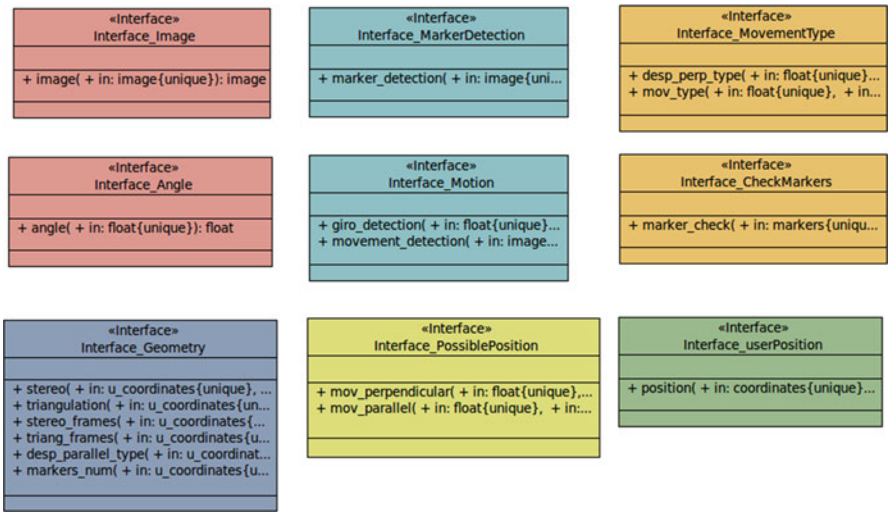

«Interface»
Interface_Image
+ image(+ in: image{unique}): image
«Interface»
Interface_MarkerDetection
+ marker_detection(+ in: image{uni...
«Interface»
Interface_MovementType
+ desp_perp_type(+ in: float{unique}...
+ mov_type(+ in: float{unique}, + in...
«Interface»
Interface_Angle
+ angle(+ in: float{unique}): float
«Interface»
Interface_Motion
+ giro_detection(+ in: float{unique}...
+ movement_detection(+ in: image...
«Interface»
Interface_CheckMarkers
+ marker_check(+ in: markers{uniqu...
«Interface»
Interface_Geometry
+ stereo(+ in: u_coordinates{unique}, ...
+ triangulation(+ in: u_coordinates{un...
+ stereo_frames(+ in: u_coordinates{...
+ triang_frames(+ in: u_coordinates{u...
+ desp_parallel_type(+ in: u_coordinat...
+ markers_num(+ in: u_coordinates{u...
«Interface»
Interface_PossiblePosition
+ mov_perpendicular(+ in: float{unique},...
+ mov_parallel(+ in: float{unique}, + in:...
«Interface»
Interface_userPosition
+ position(+ in: coordinates{unique}...

Fig. 8.6 Functional view: interfaces

As shown in Fig. 8.9, the PDM describes the distribution of the elements, the processors Cortex-A7 (proc0-proc4) and processors Cortex-A15 (proc5-proc7) connected to the 2GB LPDD3 RAM (RAM) through a AXI/AHB bus (main_bus), the two bridges (bridge0 and bridge1) and the two AMBA buses (bus).

Finally, the platform specific model (PSM) which defines the mapping of the functional components in the platform is captured in the Architectural View. It describes how the functional components are mapped onto the available HW resources. Thus, during design-space exploration, several PSMs can be analyzed and evaluated. From the result of the functional and extra-functional performance analysis, the most appropriate PSM can be selected and implemented [16].

This model specifies the different allocation of the system, which are: application components-memory partitions and memory partitions-HW/SW platform resources. So firstly, the components described on the Application View have to be associated with the memory partitions. There will be as many memory partitions as executables in the application system; in our case, the positioning system has just one executable (Fig. 8.10).

The last step is to allocate this memory partition with the HW/SW resources defined in the HW and SW Views (Fig. 8.11).

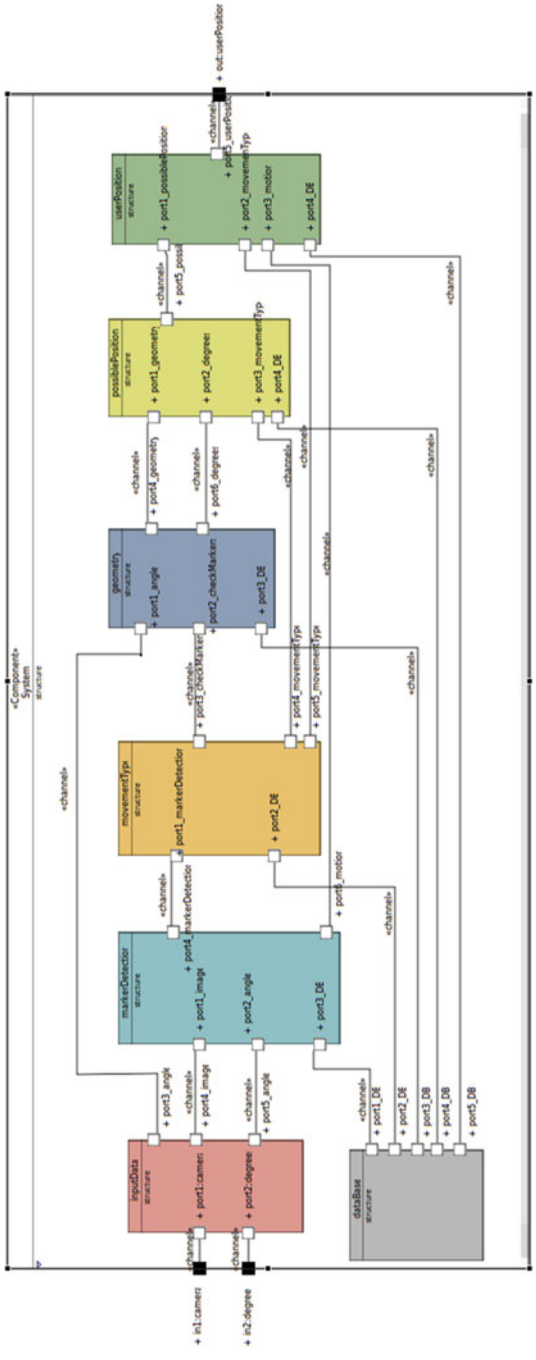


Fig. 8.7 Application view: component structure

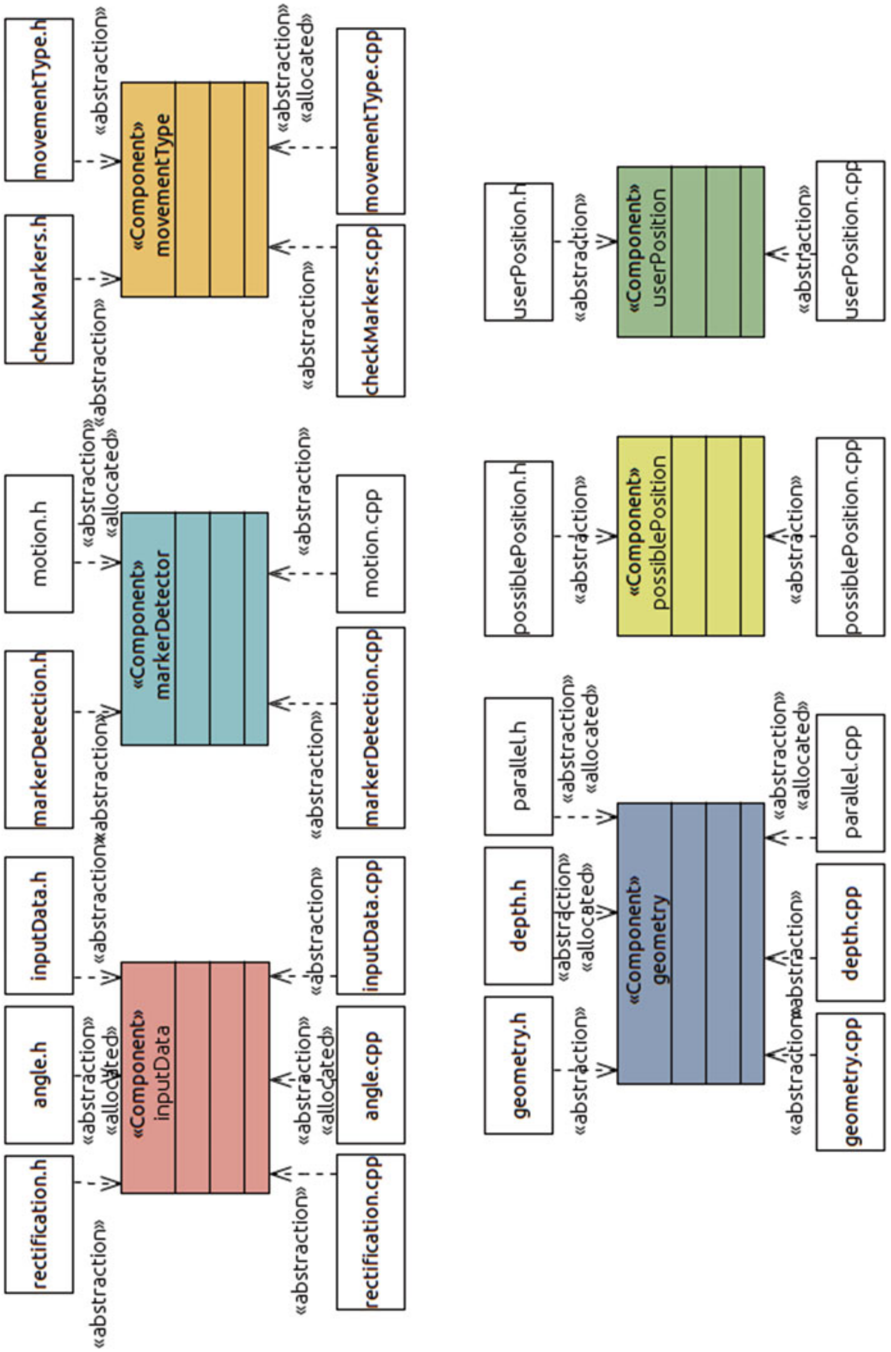


Fig. 8.8 Application view: association of C/C++ files to components

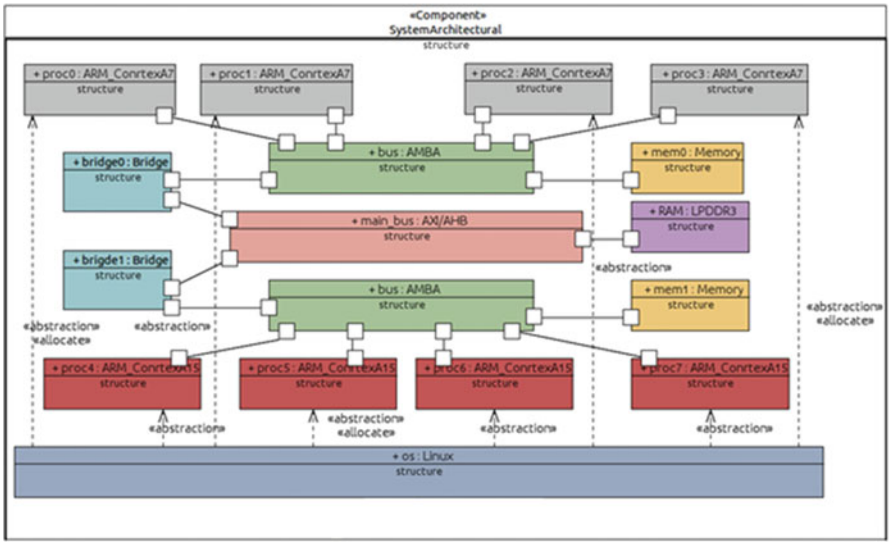


Fig. 8.9 HW/SW platform view

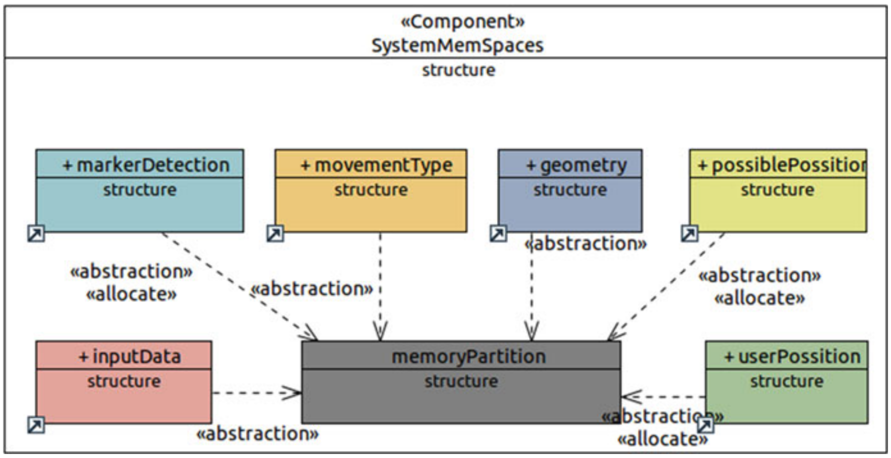


Fig. 8.10 Application component-memory partition allocation

8.3 Positioning Algorithms

This section describes the functional (SW) components of the Application View and their functionality (either provided or required) as described in the Functional View of the UML/MARTE model. The global behavior emerging from the close interaction of the behaviors of all the system components will ensure the detection and analysis of the markers and, based on this information, the positioning of objects or individuals in a 3D environment.

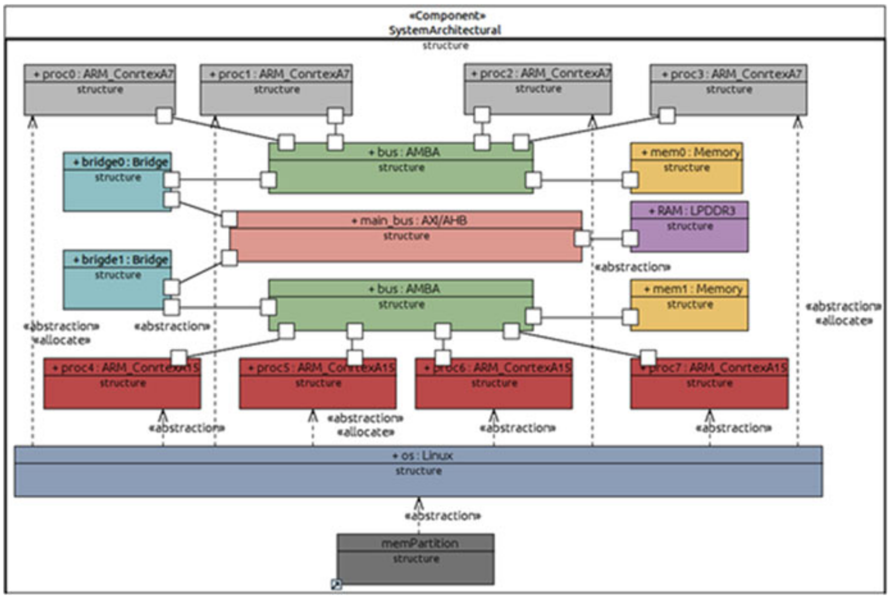


Fig. 8.11 Architectural view

8.3.1 *Input Data*

The *inputData* component is in charge of getting the view angle from the gyroscope and the images from the stereo cameras. The main functions in this component are the following (Fig. 8.12):

- Camera: Capture right and left images from the environment with a stereo camera. The cameras are installed on virtual reality glasses, such as Oculus Rift. The algorithm captures images from the camera and transforms the pair of images into rectified and undistorted images by compensating for nonlinear effects of the lens, such as radial and tangential lens distortion. Then, image quality is improved by removing sensor noise, in order to reduce to a minimum the mismatches between left and right images.
- angleMeasurement: Obtains rotation angles form a gyroscope or an electronic compass. This parameter is captured from the integrated gyroscope installed on the Oculus Rift Glasses.

8.3.2 *Marker Detection*

In the *markerDetection* component, the markers are found in the images captured and the movement of the object analyzed (Fig. 8.13).

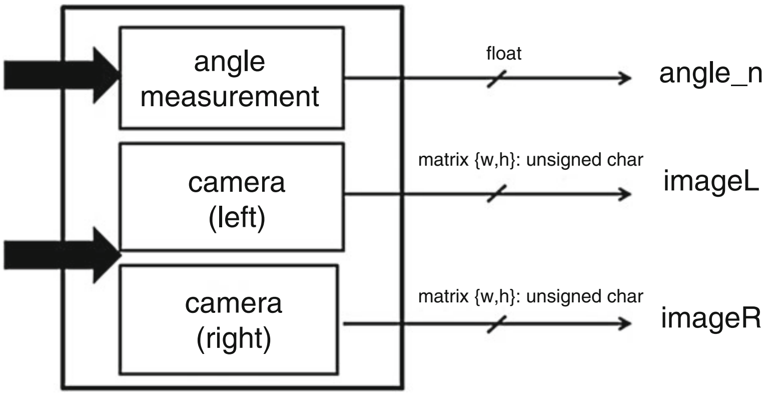


Fig. 8.12 Input data component

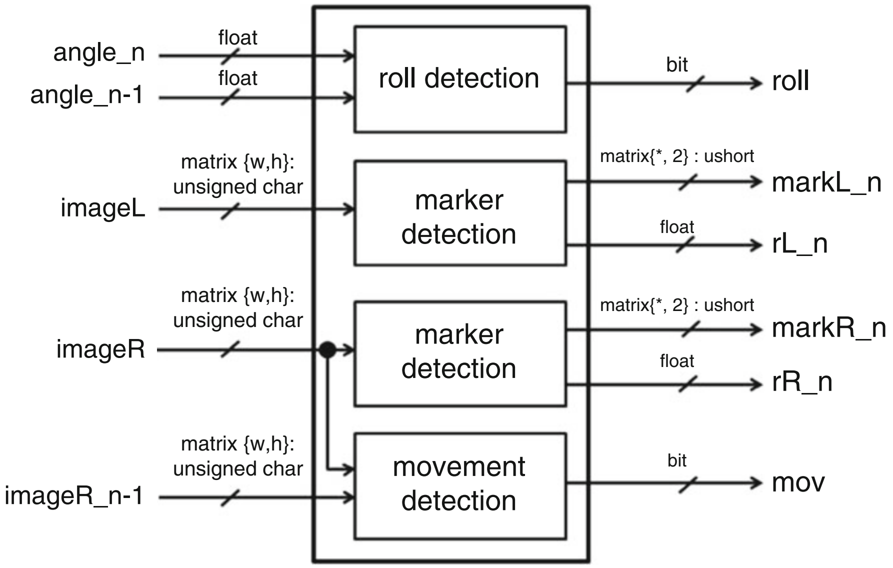


Fig. 8.13 Marker detection component

- markerDetection: This first task obtains the image coordinates of the reference makers and their radii *(u, v, r)*. As explained above, the reference markers are composed of an LED light and a contrast surface. Thus the method used to detect them is based on the search for brightest pixels' contours on the image and the verification of dark areas around them. Going into more detail, firstly, the algorithm finds pixels with color range from [205, 205, 205] to [255, 255, 255], which correspond to high luminance values, and then filters the binary image to remove non-significant bright regions by using a smooth filter or a combination of erode and dilate processes. Once the processed image is obtained, the next step is to identify the contours of the bright areas. This process is based on the algorithm presented in the article "Topological Structural Analysis of Digitized Binary Images by Border Following." In order to reduce the vertices or points which characterize the selected area, each contour is stored as a vector of points, represented as a polygon. Then, it is possible to calculate the minimal up-right bounding rectangle for the point set. Two techniques are applied to delete false positives, the first one takes into account the shape of the markers, by excluding those rectangular areas with large differences between width and height values. The second one considers the intensity difference between the light source and the contrast surface. It searches those regions with an important contrast between the selected brightest zone and its neighbor area, by obtaining the average intensity value of all pixel points in the interest area and around this area.

- rollDetection: A second function checks the value of the gyroscope to find out if the user/object has turned. It compares the angle value at the current time with the previous time instant. If they are different, it means that the user has turned.
- movementDetection: A third task detects changes between the current frame and the previous frame, by subtraction of the pixels of one, to find out if something in the image has moved. In the event of big discrepancies between the two images, this will imply that the user/object has moved in the environment.

8.3.3 Movement Type

Once the markers are detected, the next step is to recognize the type of movement performed by the user. This information is needed by the rest of the components of the positioning algorithm, and is computed by the *movementType* component and their services (Fig. 8.14):

- movementType: Identify the type of movement, whether it is perpendicular (from front to back or vice versa) or whether it is parallel (from right to left and vice versa). For this purpose, the previous and current marker radii are considered. When they are similar, it means that the user has moved in parallel, otherwise the user has moved perpendicularly.
- despy_perp_type: Taking into account the last option, it is possible to characterize the perpendicular movement; when the current radius is bigger than the previous one, the user has approached the markers, otherwise, the user has moved away from the markers.
- marker_check: Verify current marker coordinates by comparing them with the previous image region. There must be agreement between them, if not, the marker which is in a different position to the expected one will be identified as a false positive and removed from the marker array.

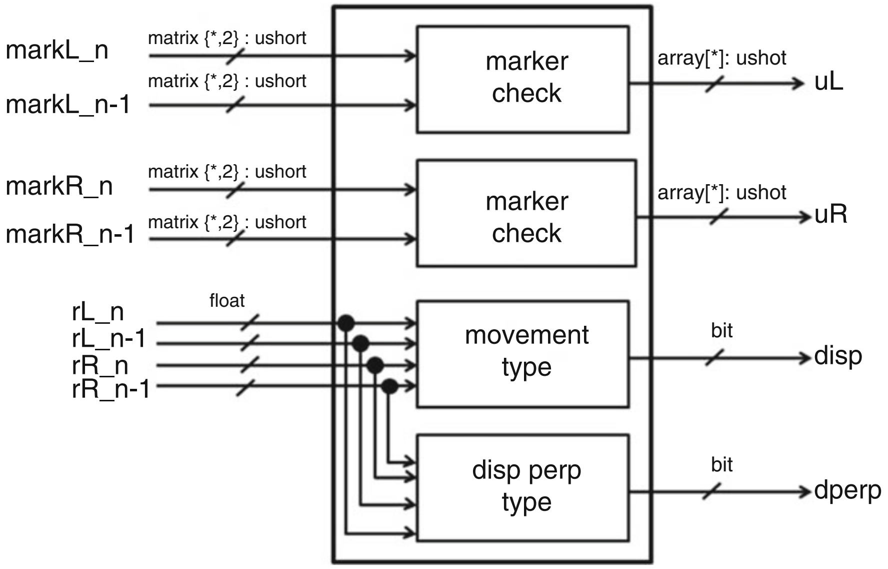


Fig. 8.14 Movement type component

8.3.4 Geometry Algorithms

The *geometry* component is in charge of the estimation of the user/object position applying the positioning algorithms. Thus, they are one of the main components of the system. To place the target user in the environment where the system is used, it is necessary to know the following information:

- the image coordinates (*u, v*) and radius of each marker captured by the stereo camera,
- the δ value (in degrees) of the target user rotation, returned by the angle measuring device (gyroscope) at the time of the stereo image capture; and
- the data stored in memory such as: previous position of the object, actual distance between markers, camera focal length, aperture angle of the camera, distance between cameras or baseline, previous image frame, previous marker radius, previous marker position vectors, and previous rotation angle.

The geometrical analysis to be performed has two parts, each one related to each of the movement components, perpendicular and parallel. In both cases, two different situations have to be considered. In the first one, only a marker is detected and as a consequence stereo-vision techniques have to be used. In the second, several markers are detected.

8.3.4.1 Perpendicular Movement

Geometry Interface, Stereo Operation: Single Marker

This case represents a perpendicular movement of the target object when only a single marker is detected by the stereo camera. It is necessary to make use of stereo-vision techniques. The following parameters are needed:

- the values of baseline (B) and the focal length distance (f) of the stereo camera
- the rotation angle (δ) of the target object
- preceding target object position (x_{n-1}, y_{n-1})
- preceding distance between the marker and the camera ($\mathrm{L}_{\mathrm{marker}\_n-1}$) (Figs. 8.15 and 8.16).

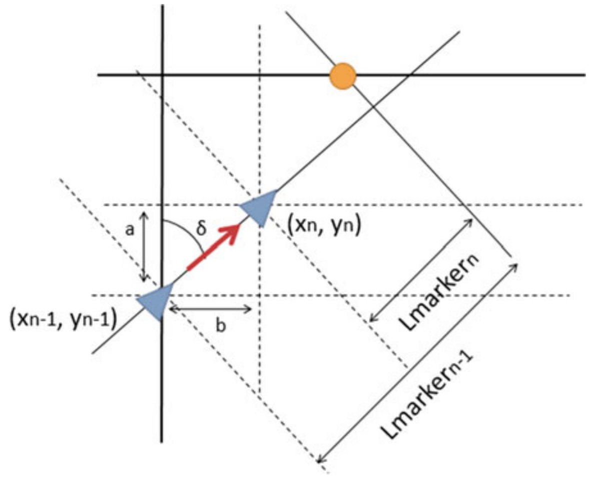


Fig. 8.15 Perpendicular movement with one marker

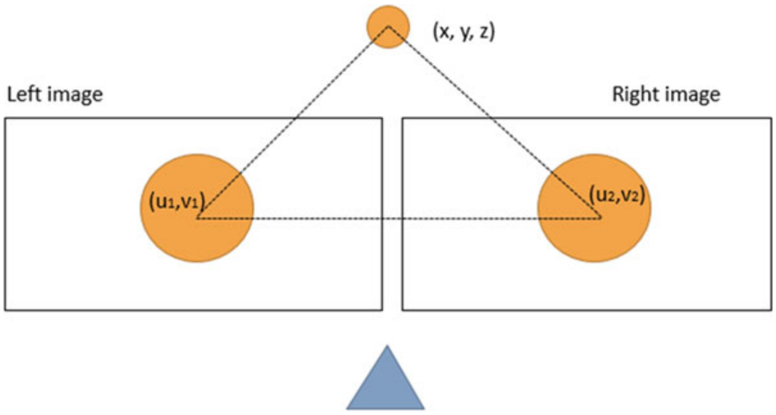


Fig. 8.16 Epipolar geometry

To transform the image coordinates (u, v) to the depth between the marker and the camera ($L_{\text{marker\_stereo\_}n}$), projective geometry is applied at the current time n according to the following equation (where u_L and u_R are the parallel left and right image coordinates, respectively, which are rectified and undistorted):

$$L_{\text{marker\_stereo}_n} = \frac{\text{baseline} \times \text{focal\_length}}{\text{disparity}} = \frac{B \times f}{u_L - u_R}$$

Geometry Interface, Triangulation Operation: Two or More Markers

When two or more markers are detected in the image it is possible to apply linear triangulation. In this case, the following parameters are required:

- the real distance (d/m) between markers,
- the rotation angle (δ),
- the camera opening angle (2φ),
- the image pixels ($A \times B$),
- preceding target object position (x_{n-1}, y_{n-1}),
- preceding distance between the marker and the camera ($L_{\text{marker\_}n-1}$) (Figs. 8.17 and 8.18).

Once the parallel image marker coordinates (u) are known, the distance in pixels between them ($q = u_2 - u_1$), which in the real world is equal to d/m m. Therefore, the actual distance, L_{marker} (n), between the target and one of the markers at the current instant n is

$$L_{\text{marker\_triang}_n} = \frac{\left|\frac{A}{2} - u_1\right| \times \frac{d}{m} \times \cos\ \delta \times \frac{1}{|u_2 - u_1|}}{\tan\left(2\varphi \times \left|\frac{A}{2} - u_1\right|\right)}$$

The following figure shows how these algorithms are implemented in the system (Fig. 8.19).

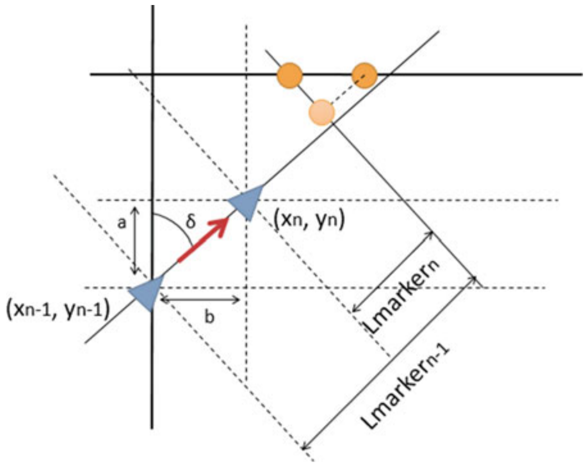
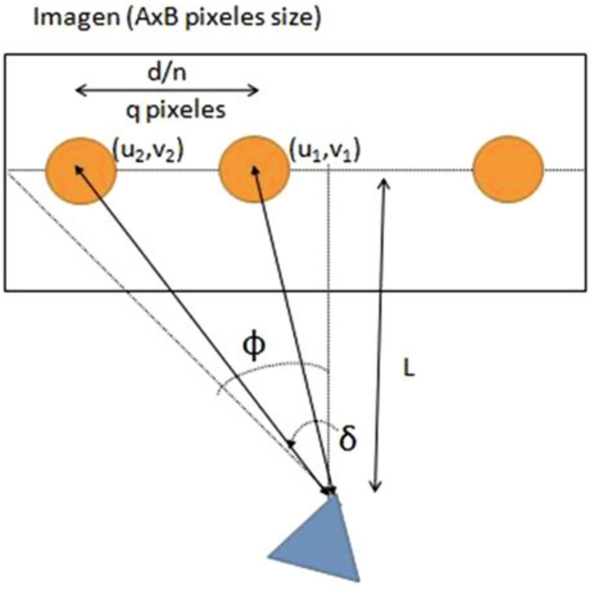


Fig. 8.17 Perpendicular movement with two markers

Fig. 8.18 Triangulation



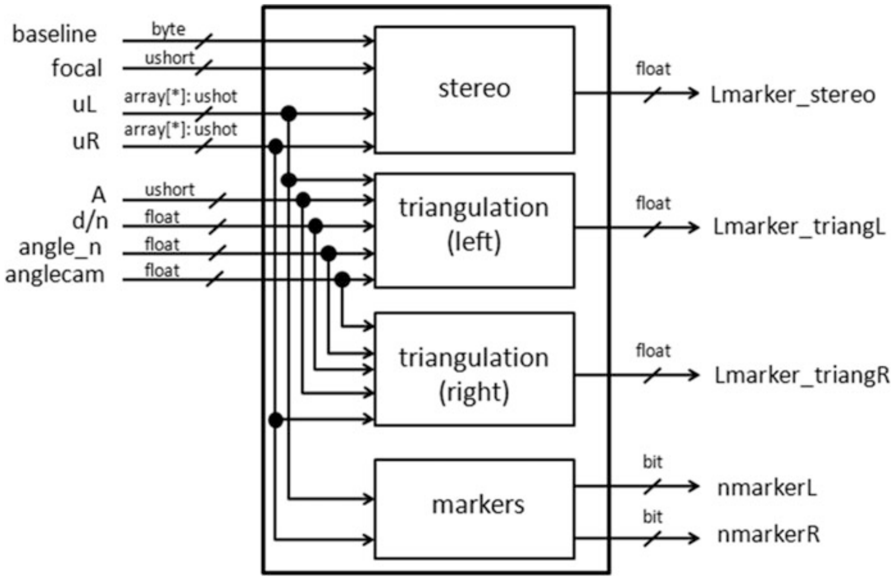


Fig. 8.19 Perpendicular movement in the geometry algorithm component

8.3.4.2 Parallel Movement

Geometry Interface, Stereo_Frames Operation: Single Marker

As with perpendicular movement, an algorithm based on stereo geometry is applied, but instead of using two images of the same instant taken from two different angles (each camera), it uses two pictures of the previous instant with the same perspective. The parameters required for this configuration are

- the focal length (f) of the stereo camera
- the rotation angle (δ)
- preceding target object position (x_{n-1}, y_{n-1})
- preceding distance between the marker and the camera (L_{marker_n-1})
- preceding u-coordinate of the marker (Figs. 8.20 and 8.21)

Knowing the parameters presented above, the parallel shift (D) can be calculated by the target user according to the following expression:

$$D_{\text{stereo}} = \frac{L_{\text{marker}_{n-1}} \times |u_{n-1} - u_n|}{f}$$

Geometry Interface, Triang_Frames Operation: Two or More Markers

This last situation is related to the case where there is more than one detected marker in the image. A similar algorithm is used to the triangulation one previously explained in the case of a perpendicular movement of the user with many markers detected, but in this case on two images captured consecutively in time by the same image sensor. In this case, it is necessary to know the following values:

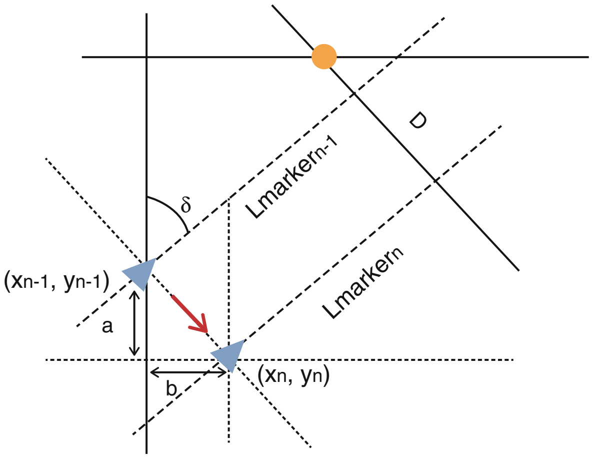


Fig. 8.20 Parallel movement with one marker

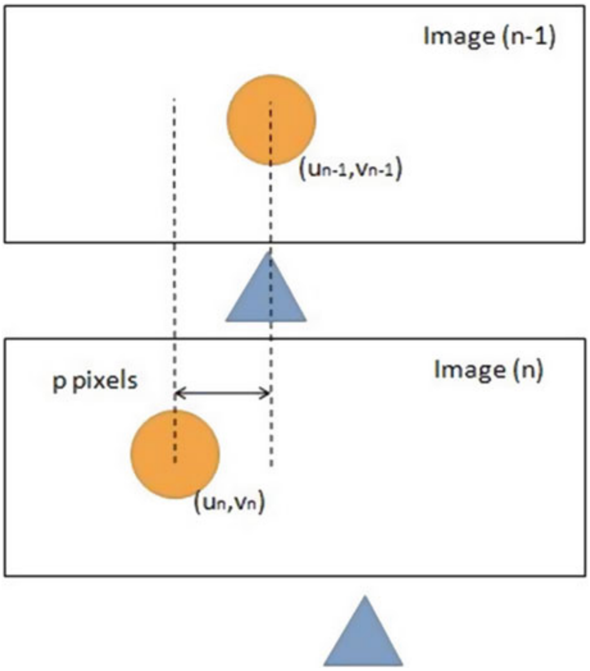


Fig. 8.21 Stereo between consecutive frames

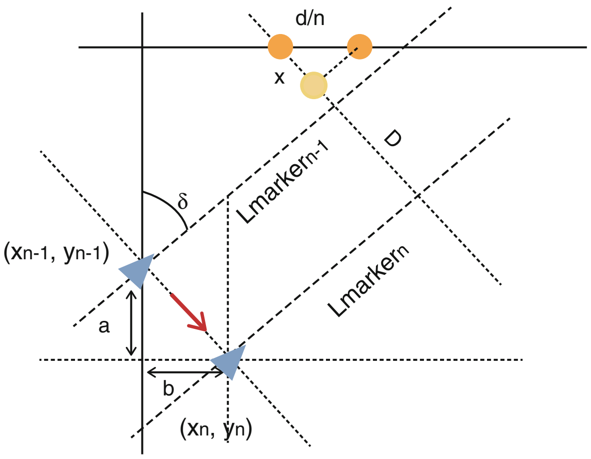


Fig. 8.22 Parallel movement with two markers

- the rotation angle (δ)
- preceding target object position (x_{n-1}, y_{n-1})
- preceding *u*-coordinate of the marker
- the real distance (*d/m*) between markers (Figs. 8.22 and 8.23)

Knowing the real distance between markers and the pixels between them, at the current instant and its previous instant, the distance that the target object has moved can be extrapolated:

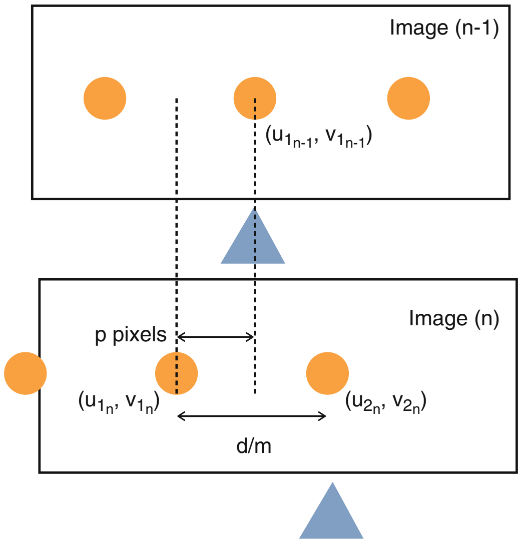


Fig. 8.23 Triangulation between consecutive frames

$$D_{\text{triang}} = \cos(\delta) \times \frac{d}{m} \times \frac{\left|u_{1_{n-1}} - u_{1_n}\right|}{u_{2_n} - u_{1_n}}$$

The figure shows the last algorithms and how they are implemented in the system (Fig. 8.24).

Others Functions

The geometry algorithm component also implements other important features:

- geometry interface, markers_num operation: Classifying the number of markers in each image. When calculating the user position, the algorithm needs to know whether there is a single marker or more than one marker. This parameter is easy to compute by monitoring the length of the marker coordinate array.
- geometry interface, desp_parellel_type operation: Identifying the type of parallel displacement (from right to left or vice versa). The parallel movement is characterized by comparing the current parallel image coordinates (u-coordinates) of the marker array with the previous markers. If the new ones have higher values, the user has moved to the left, otherwise, the user has moved to the right.

8.3.4.3 Possible Position

The *possiblePosition* component compares different distance values calculated in the last stage for each type of movement and selects the correct ones, considering the

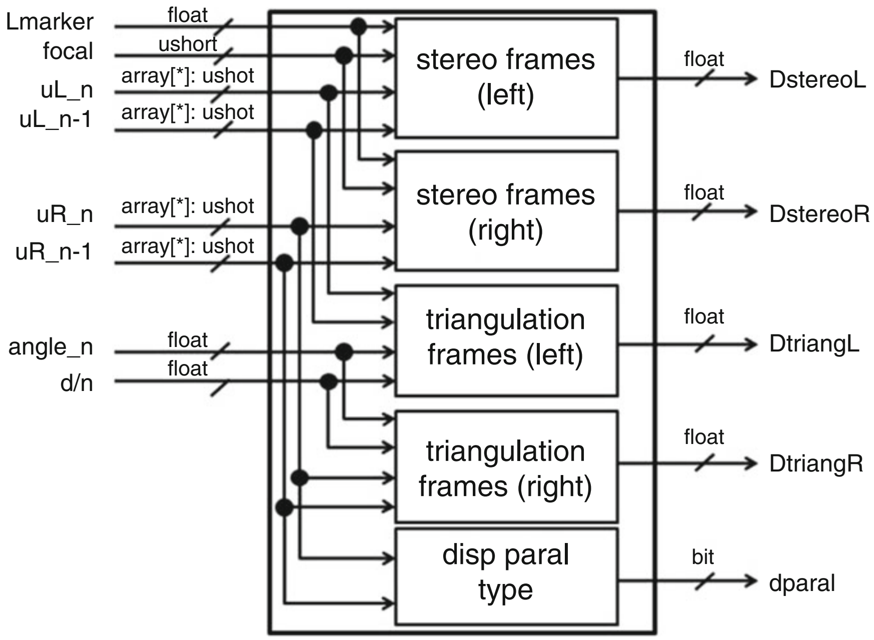


Fig. 8.24 Parallel movement in geometry algorithm component

type of perpendicular or parallel displacement and the number of markers detected in the image. Again, different solutions are applied depending on whether there is only one marker or more than one in the image.

Possibleposition Interface, Mov_Perpendicular Operation: Perpendicular Movement

Single Marker

Once the distance $L_{\text{marker\_stereo\_}n}$ is calculated, it is possible to obtain the position of the target in the environment. As it has been displaced perpendicularly, all that has apparently changed is its y-coordinate, but it is necessary to consider the angle of rotation (δ) for absolute coordinates. The coordinates (x, y) at the present moment are equal to the coordinates in the previous instant (x_{n-1}, y_{n-1}) plus the new movement.

If $r_{n-1}<r_n$	$\{$	$x_n = x_{n-1} + \sin(\delta) \times \lvert L_{\text{marker}_{n-1}} - L_{\text{marker\_stereo}_n} \rvert$
		$y_n = y_{n-1} + \cos(\delta) \times \lvert L_{\text{marker}_{n-1}} - L_{\text{marker\_stereo}_n} \rvert$
If $r_{n-1}>r_n$	$\{$	$x_n = x_{n-1} - \sin(\delta) \times \lvert L_{\text{marker}_{n-1}} - L_{\text{marker\_stereo}_n} \rvert$
		$y_n = y_{n-1} - \cos(\delta) \times \lvert L_{\text{marker}_{n-1}} - L_{\text{marker\_stereo}_n} \rvert$

Two or More Markers

When the distance to the marker is calculated, and having the previous $n-1$ distance, the new distance covered is the difference between them. From this value and the previous position of the target (x_{n-1}, y_{n-1}), its new coordinates (x_n, y_n) are

If $r_{n-1} < r_n$	$\{$	$x_n = x_{n-1} + \sin(\delta) \times \lvert L_{\text{marker}_{n-1}} - L_{\text{marker\_triang}_n} \rvert$
		$y_n = y_{n-1} + \cos(\delta) \times \lvert L_{\text{marker}_{n-1}} - L_{\text{marker\_triang}_n} \rvert$
If $r_{n-1} > r_n$	$\{$	$x_n = x_{n-1} - \sin(\delta) \times \lvert L_{\text{marker}_{n-1}} - L_{\text{marker\_triang}_n} \rvert$
		$y_n = y_{n-1} - \cos(\delta) \times \lvert L_{\text{marker}_{n-1}} - L_{\text{marker\_triang}_n} \rvert$

Possibleposition Interface, Mov_Parallel Operation: Parallel Movement

Single Marker

As soon as the displacement D in meters of the target user is known, it is possible to obtain its actual coordinates, which depend on its position in the previous time $n-1$ and the type of movement (left or right):

If $u_{n-1} > u_n$	$\{$	$x_n = x_{n-1} - \cos(\delta) \times D_{\text{stereo}}$
		$y_n = y_{n-1} + \sin(\delta) \times D_{\text{stereo}}$
If $u_{n-1} > u_n$	$\{$	$x_n = x_{n-1} + \cos(\delta) \times D_{\text{stereo}}$
		$y_n = y_{n-1} - \sin(\delta) \times D_{\text{stereo}}$

Two or More Markers

As in the case of a single marker, once the displacement (D) is known, the actual coordinates of the target object can be obtained:

If $u_{n-1} < u_n$	$\{$	$x_n = x_{n-1} - \cos(\delta) \times D_{\text{triang}}$
		$y_n = y_{n-1} + \sin(\delta) \times D_{\text{triang}}$
If $u_{n-1} > u_n$	$\{$	$x_n = x_{n-1} + \cos(\delta) \times D_{\text{triang}}$
		$y_n = y_{n-1} - \sin(\delta) \times D_{\text{triang}}$

The following is the Possible Position component (Fig. 8.25).

8.3.4.4 User Position

The last step, *userPosition* component, obtains the target object or individual position in the 3D environment, represented with the service *position*. It considers

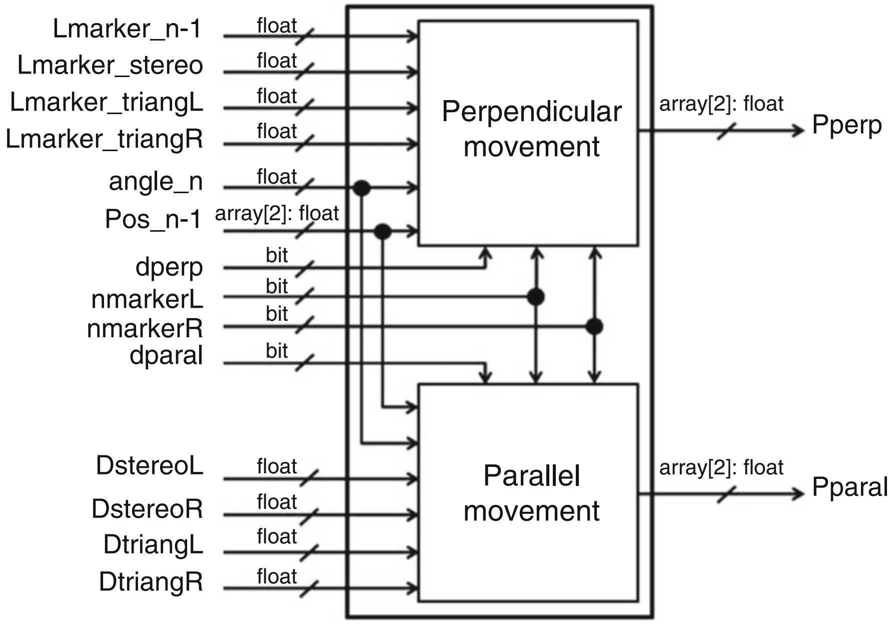


Fig. 8.25 Possible position component

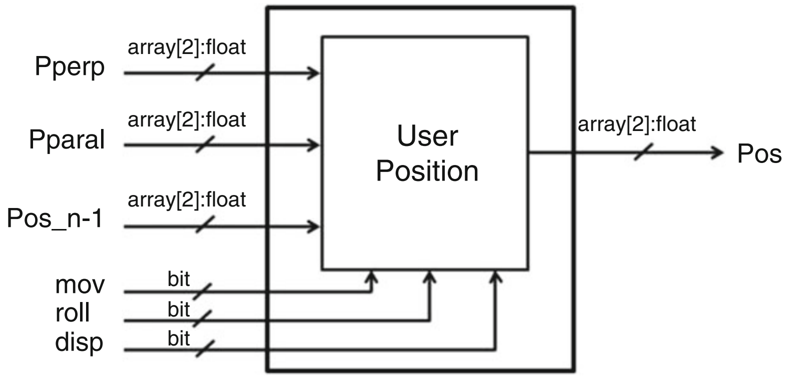


Fig. 8.26 User position component

the data received from the last stage, *Pperp* and *Pparal*, and the previous position. The final position is selected from one of those three, taking into consideration the values of movement type and whether there has been displacement or rotation (Fig. 8.26).

8.4 Synthesis

Once the complete platform model has been created and all the functionalities are implemented, simulation and synthesis are the last steps. Firstly, it is necessary to generate the XML files from the model and the wrappers files from these XML files. Then it is possible to generate the makefiles and compile the complete system. Finally the generated binary files are ready to run into the development board.

The system model design includes the evaluation in seconds or data/second of each component. The idea is to have a system working properly, where the user does not notice the computational charge. In our case, we have three important limitations:

- the minimum delay for the human sensory system using a VR glasses
  – 33 ms or less will provide the minimum level of latency deemed acceptable, in frequency terms is 30 frames per second
- the minimum resolution to detect the markers in a space where the dimensions between the user and the marker could be at least 10 m
  – 640×480 pixels/image
- the minimum delay for the human sensory to locate someone on an environment
  – 250 ms

These temporal restrictions have been included in the model. It will be said that the model fidelity has been satisfied by the implementation whenever the restrictions has been satisfied.

	Components	Input data	Marker detection	Movement type	Geometry
	Temporal restriction (ms)	<167	<80	<2	<0.5
PC	Measured latency (ms)	81.104	14.222	0.088	0.010
Board	Measured latency (ms)	137.381	59.154	0.114	0.019

The positioning system has not been yet completely implemented as the two last components of the application are not yet finished. However, a first approximation of the model and a time analysis is done, which could be compared with the ideal data. The goal is to get the response time results in order to know when it is necessary to parallelize or optimize the code. And, if necessary, change the hardware, by using a camera with USB 3.0, which has ten times more MB rate than a camera with USB 2.0, or by using a different development board.

8.5 Conclusions

The advances during the last years in video display, video processing, and graphic SW technologies have led to the emergence of modified reality systems. Three modified reality systems can be distinguished. First, those systems adding information to reality, called augmented reality systems. Second, those systems creating fully artificial worlds, called virtual reality systems. In between, there are systems able to take real and virtual images and create a synthetic new image by combining both. We have called them "recreated reality."

One of the main problems in all these technologies is accurate positioning of the subject. Many different technologies have been proposed to date. Some of them are valid only in small spaces. Others only work indoor or outdoor. The University of Cantabria as a consequence of its participation to the Artemis CopCams project has developed a positioning system able to overcome most of the drawbacks found by other competing alternatives. In this chapter, the technique has been described and its UML/MARTE model outlined. At the end of the project, in September 2016, a prototype is forecast that will be able to assess the actual capabilities of the proposed positioning system.

UML/MARTE has proven to be a powerful means to model this high-performance video processing system. From the UML/MARTE model the designer can decide about the system, detect potential architectural problems, fully specify the different components and based on this, start the platform-independent code development. Once the code is ready, the model can be simulated and finally, synthesized (www.essyn.com). By using the SW synthesis tool eSSYN, it is possible with very fast turn-arounds to verify if the temporal restrictions imposed by the model are satisfied by the implementation, that is, to what extend the fidelity to the model has been achieved.

References

1. BeAnotherLab, Retrieved from http://www.themachinetobeanother.org/ (2014)
2. J. Chen, *Guide to Graphics SW Tools* (Springer, NewYork, 2008)
3. Google, Retrieved from Google (2016). https://www.google.com/get/cardboard/ (2016)
4. F.P. Herrera, The COMPLEX Methodology for UML/MARTE modeling and design-space exploration of embedded systems. J. Syst. Archit. **60**(1), 55–78 (2014). Elsevier
5. R. Kumar, S. Samarasekera, T. Oskiper, Method and apparatus for 3D spatial locazation and traking of objects using active optical illumination and sensing. WO Patent 2013/120041 A1 (2013)
6. M. Maeda, T. Ogawa, K. Kiyokawa, H. Takemura, Tracking of user position and orientation by stereo measurement of infrared markers and orientation sensing. *IEEE, 8th International Symposium on Wearable Computers* (2004)
7. Microsoft, Retrieved from https://www.microsoft.com/microsoft-hololens/en-us (2016)
8. A. Mossel, H. Kaufmann, Wide area optical tracking in unconstrained indoor enviroments. *IEEE, 23rd International Conference on Artificial Reality and Telexistence*, (2013)
9. L. Naimark, E. Foxlin, Fudicial detection system. US Patent 7,231,063 B2 (2007)

10. Y. Nakano, K. Izutsu, K. Tajitsu, K. Kai, T. Tatsumi, *Kinect Positioing System (KPS) and its Potential Applications.* International Conference on Indoor Positioning and Indoor Navigation (2012)
11. OMG, Modelling and analysis of real-time embedded systems, Version 1.1. OMG. Available from http://www.omgmarte.org (2012)
12. P. Peñil, UML-Marte methodology for heterogenius system design. Microelectronics Engineering Group, TEISA Dpt., University of Cantabria (2014)
13. H. Posadas, P. Peñil, A. Nicolás, E. Villar, Automatic synthesis of embedded SW for evaluating physical implementation alternatives from UML/MARTE models supporting memory space separation. Microelectr. J. **45**(101), 281–1291 (2014)
14. Samsung, Retrieved from http://www.samsung.com/au/news/local/world-first-live-streaming-virtual-reality-birth-using-samsung-gear-vr (2015)
15. D.C. Schmidt, Model-driven engineering. IEEE Computer **39**(2), 25–31 (2006)
16. C.F. Silvano, *Multi-Objective Design-Space Exloration of Multiprocessor Soc Architecture* (Springer, NewYork, 2011)
17. Tom's Guide, Retrieved from http://www.tomsguide.com/us/best-ar-glasses,review-2804.html (2016)
18. UML, *Unified Modelling Language.* OMG. Available from http://www.omg.org/spec/UML
19. E. Villar, P. Martínez, F. Alcalá, P. Sánchez, V. Fernández, Método y sistema de localización espacial mediante marcadores luminosos para cualquier ambiente. P.N.ES-2543038-A1 (2014)
20. C. Wang, C. Chen, *RFID-based and Kinect-based Indoor Positiong System.* 4th Int. Conference on Wireless Communications, VehicularTechnology, Information Theory and Aerospace & Electronic Systems (VITAE) (2014)

Index