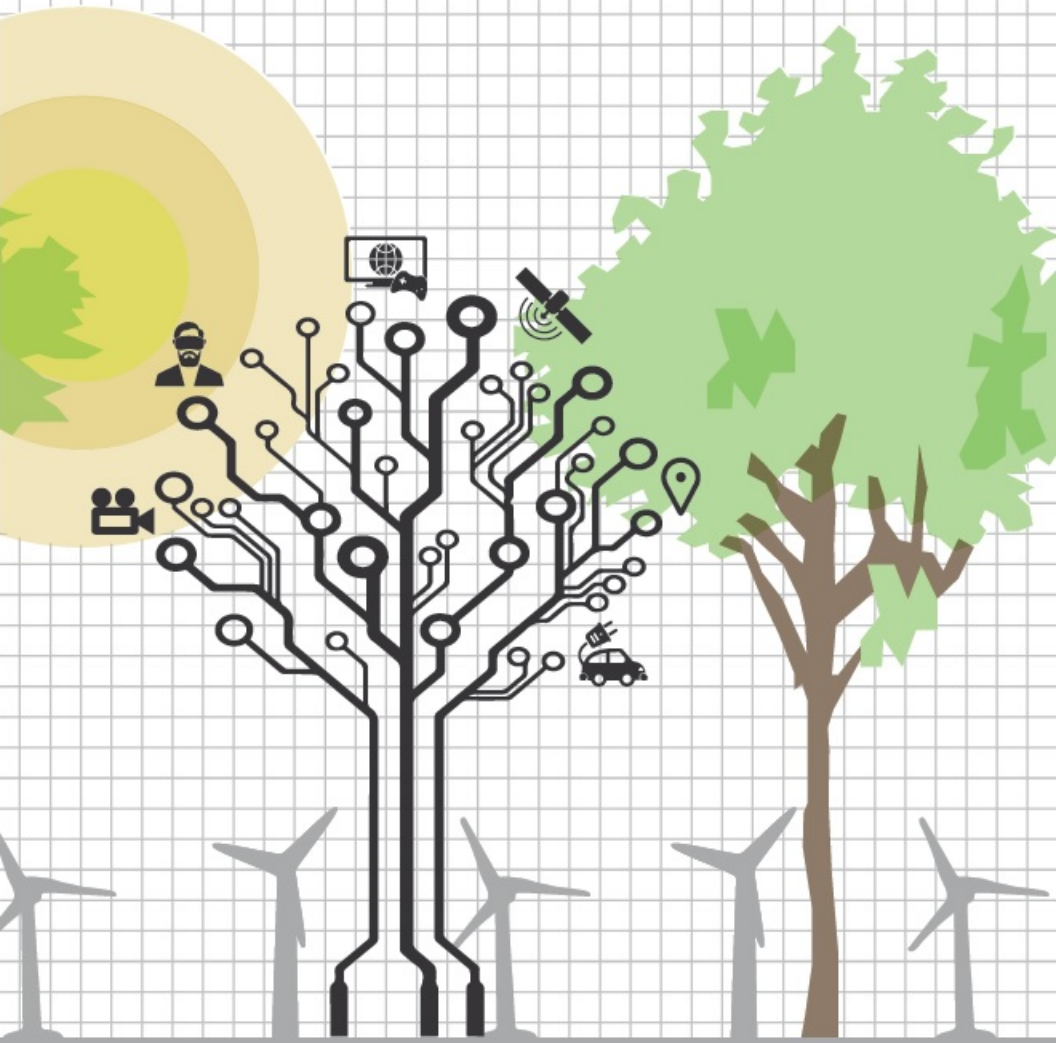


GREEN COMPUTING

Efficient Energy Management of Multiprocessor Streaming Applications via Model Checking

Waheed Ahmad

Green Computing: Efficient Energy Management of Multiprocessor Streaming Applications via Model Checking

Waheed Ahmad

Graduation committee:

Chairman:	Prof. dr. P.M.G. Apers
Promotor:	Prof. dr. J.C. van de Pol
Co-promotor:	Dr. M.I.A. Stoelinga

Members:

Prof. dr. ir. B.R.H.M. Haverkort	University of Twente
Dr. ir. J.F. Broenink	University of Twente
Prof. dr. K.G.W. Goossens	Eindhoven University of Technology
Prof. dr. W. Yi	Uppsala University
Dr. ir. P.K.F. Hölzenspies	Facebook, London

CTIT Ph.D. Thesis Series No. 16-418
Centre for Telematics and Information Technology
University of Twente, The Netherlands
P.O. Box 217 – 7500 AE Enschede

IPA Dissertation Series No. 2017-02
The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

The work in this thesis is conducted within Self Energy-Supporting Autonomous Computation (SENSATION) project (318490) supported by European Commission.

ISBN 978-90-365-4290-6
ISSN 1381-3617 (CTIT Ph.D. Thesis Series No. 16-418)
Available online at `https://doi.org/10.3990/1.9789036542906`

Typeset with LaTeX
Printed by Ipskamp Drukkers
Cover design © by Annelien Dam

GREEN COMPUTING: EFFICIENT ENERGY MANAGEMENT OF MULTIPROCESSOR STREAMING APPLICATIONS VIA MODEL CHECKING

DISSERTATION

to obtain
the degree of doctor at the University of Twente,
on the authority of the rector magnificus,
Prof. dr. T.T.M. Palstra,
on account of the decision of the graduation committee,
to be publicly defended
on Thursday, April 13th, 2017 at 12:45 hrs.

by

Waheed Ahmad

born on 4 July 1987
in Lahore, Pakistan

This dissertation has been approved by:

Prof. dr. J.C. van de Pol (promotor)
Dr. M.I.A. Stoelinga (co-promotor)

To Prof. dr. Abdus Salam (in memoriam)
Nobel Laureate in Physics 1979

Acknowledgements

I still remember when I came to Enschede for the PhD position interview, four and half years ago. With a background in electronics engineering and knowing little of the field of formal methods, I was a bit nervous. At the end of the interview, after some discussion between Jaco and Mariëlle, I was offered this position, for which I am really thankful. In other words, thank you for offering four wonderful and unforgettable years.

Jaco, in your role as a promotor, you had a big part in shaping my academic skills. Your eye for technical details spotted the mistakes that I may have otherwise missed. Your rigorous mathematical analysis and insights during meetings and lunch talks helped me later in formulating the problem statements of papers in a clear and concise way. I was mostly impressed by your ability to grasp new ideas and concepts in a short space of time. For our ISoLA paper, you learned the whole concept of controller synthesis using UPPAAL STRATEGO in a short time, while it took me so long. Same was the reason for asking your help in finishing Chapter 8. I was also impressed by your planning skills. During the SENSATION project, your insistence on planning everything ahead helped in delivering all deliverables in a timely fashion.

Mariëlle, as my daily supervisor, I have learned the most from you. I have not seen/heard any other researcher who is so good at scientific writing. You always taught me to have clean theory, and to use simple and concise structures. I always used to think that you were better suited to being a journalist than a researcher. I also learned from you the importance of building a bridge between research and its impact on society. As my boss, you gave me complete freedom to work on my own ideas, which will help me to be an independent researcher in the long run. You are full of energy all the time. Besides your super busy schedule, it's unbelievable how you make time for your kids and family and take care of them. My deepest gratitude to you and Jaco for supporting me in the best possible way, when my daughter was in the hospital. Thank you for facilitating what turned out to be the most difficult period of my life to a great extent. Hartelijk bedankt.

In addition to my supervisors, I would also like to thank Robert de Groote for helping me to understand the vast jungle of dataflow. Thank you for your support and collaboration on our ACSD'14 paper, even though you are not a big fan of model checking. I would also like to thank Philip Hölzenspies for taking interest in my work. Your keenness to learn new concepts and ideas always inspired me. Your areas of speciality range from computer architectures and dataflow applications, to programming languages and model checking, and

working and collaborating with you allowed me to acquire some taste of these areas. You are a genuine all-rounder (in cricket terms).

An important part of my PhD experience consisted of my stay in the Formal Methods and Tools (FMT) group. Thank you for being there, for having cosy chats in Rappa, and organising movie nights and of course trainings for the Batavierenrace. Big thank you to Stefano for giving feedback on my thesis, and being my paranymph. Thank you Bugra for introducing me to the world of model-driven engineering, and giving me a place to stay during the last two months of my PhD. I will always remember our trip to Italy. Thanks to Rajesh for always willing to chat on any topic. Last but not least, my heartfelt thanks to Joke and Ida, for always lending a helping hand and taking care of administrative tasks. You are truly a blessing for FMT.

I thank the members of my committee for approving my thesis and providing helpful comments. I also thank Timon ter Braak and Kim Sunesen from Recore Systems, for providing the case study and helping with analysis.

Next, I would like to thank my friends in Enschede, especially my friends from the Pakistani Student Association. Thank you for organising great events throughout the year, in particular Basant, which always made Enschede feel like home to me.

I thank my parents and siblings for their love and support over the years to chase and achieve my dreams. I thank my father -my first teacher- for cultivating my love for science and books. I also thank my mother for teaching diligence, endurance and patience. I would also convey my deepest thanks to my in-laws for their unconditional love and support.

I saved the best for my wife, Friha. Thank you for putting up with me over the past few months when I was often busy. Thank you for taking care of me during the strenuous process of writing the thesis. Luckily, we had the opportunity to spend some amazing time together. In particular, our unforgettable trips to Tunisia, Spain, Portugal and Greece. And not to forget our daughter Sabika Noor Aulakh! Thank you for bringing colours in our lives.

Nuenen
March 2017

Abstract

Consumer electronics such as televisions, telephones, and computers have become an essential part of a human life. An important subclass of consumer electronics termed *embedded multimedia systems* deal with applications from the multimedia and Digital Signal Processing (DSP) domain executing on multiprocessors. Such applications repetitively process an input stream of indefinite length, for example a video decoder that decodes a video stream. These applications are often referred to as *streaming applications* in literature. Examples of embedded multimedia systems are mobile phones, virtual reality gaming consoles, 3D-enabled televisions, and car navigation systems. The Synchronous Dataflow (SDF) model of computation naturally captures the characteristics of streaming applications and allows design-time analysis of timing and resource utilisation.

Embedded multimedia systems have evolved significantly in recent decades, and are becoming ubiquitous in our daily lives. This trend is also giving rise to challenges such as (1) increasing energy demand leading to global warming, (2) requirement of seamless and robust performance, and (3) growing complexity of embedded multimedia systems resulting in higher development cost and longer time-to-market.

To address these challenges, we introduce several methods that combine resource and power management with scheduling decisions. As an analysis environment, we consider model checking because of its ability to generate optimal traces (schedules).

The first approach is *throughput-optimal* scheduling of SDF graphs on a given number of processors via the proven formalism of timed automata. In this work, SDF graphs along with hardware platforms are translated compositionally to timed automata. The problem of throughput optimisation is encoded as a query over timed automata. The model checker UPPAAL extracts a trace representing a throughput-optimal schedule. In this way, we can efficiently determine a trade-off between number of processors and throughput for a certain streaming application.

The second approach generates *energy-optimal* schedules of SDF graphs. The hardware architecture is decorated with novel energy management techniques like dynamic power management (DPM, switching to low power state) and dynamic voltage and frequency scaling (DVFS, throttling processor frequency). To balance flexibility and design complexity, the concept of Voltage and Frequency Islands (VFIs) is considered. It achieves fine-grained system-level power management,

by operating all processors in the same VFI at a common frequency/voltage.

In this work, we utilise priced timed automata, a model checking formalism that extends timed automata with costs, which are used to model the power consumption of processors. After SDF graphs and hardware platforms are translated to priced timed automata, the model checker UPPAAL CORA generates a trace representing an energy-optimal schedule. We demonstrate that the combination of DPM and DVFS provides an energy reduction beyond considering DVFS or DPM separately. Moreover, we show that by clustering processors in VFIs, DPM can be combined with any granularity of DVFS.

The third approach derives the Quality of Service of SDF graphs mapped on hardware platforms powered by multiple batteries. In this approach, we use hybrid automata which are an extension of timed automata with continuous variables. Furthermore, using the model checker UPPAAL SMC, we evaluate (1) system lifetime, and (2) minimum required initial battery capacities to achieve the desired application performance.

In today's agile world, there is a fierce competition that requires low development cost and short time-to-market. To achieve this purpose, an efficient modelling approach is needed which can provide modularity, extensibility and interoperability. We have developed a Model-Driven Engineering (MDE) based framework which fulfils these requirements. In this framework, we introduce the so-called *metamodels* for SDF graphs and hardware platforms. The SDF graphs and hardware platforms are translated to the model-checking domain automatically using *model transformations*.

Finally, we evaluate the performance of our approach of throughput analysis by applying it in an industrial case study of face recognition systems provided by Recore Systems, Netherlands. With this case study, the performance of our approach is validated in realistic scenarios and, thus, the problem is shown to be solvable with acceptable concessions.

Table of Contents

Introduction

Once, the access to consumer electronics devices such as televisions, telephones, and computers was a luxury shared by few. Nowadays, courtesy to the inventions in the field of electronics in the past half-century, consumer electronics devices have become ubiquitous in our daily lives. We are getting more dependent on computers to carry out our daily tasks, such as with-drawing money, scheduling an appointment, reading the news on a tablet, listening to music on a MP3 player, and watching a favourite film on DVD. According to MarketresearchReports.Biz, the Consumer Electronics market is going to witness a value of US $1.6 trillion by 2018 [MAR13].

Most of consumer electronics devices contain one or more processors to perform the required functionalities of the device. Such devices are termed *embedded systems*. An important subclass of embedded systems is known as *embedded multimedia systems*, which deal with processing multimedia information such as, data, voice, graphics, animations etc., in real-time. Examples of embedded multimedia systems are mobile phones, tablets, virtual reality (VR) enabled gaming consoles, music players, and car navigation systems. As multimedia applications inherently include continuous streams (e.g., streaming videos or audio clips), we can say that many embedded multimedia systems contain *streaming applications* [TKA02].

Embedded multimedia systems are on the rise to make their way in our everyday lives. This trend is also leading to the new challenges of getting a trade-off between performance, cost (number of processing or memory elements etc.), and energy consumption for these systems. In sections 1.1-1.3, we will investigate the societal, consumer and industrial challenges that led to this research on efficient performance and energy optimisation of streaming applications.

1.1 Challenges: A Societal Perspective

The demand for energy in both commercial and domestic environments is increasing. While our primary sources of energy are running out, the side effects of energy usage have adverse environmental effects. For example, climatologists are associating emission of greenhouse gases such as CO_2 to global warming. Figure 1.1 shows the worldwide energy consumption (TWh) from 1990-2015 [ENE16], where we can see that the energy usage per year has increased dramatically. In fact, in this century so far, the rate of rise in energy consumption is 56%, and this trend is expected to grow at a similar rate. Former U.S. Secretary

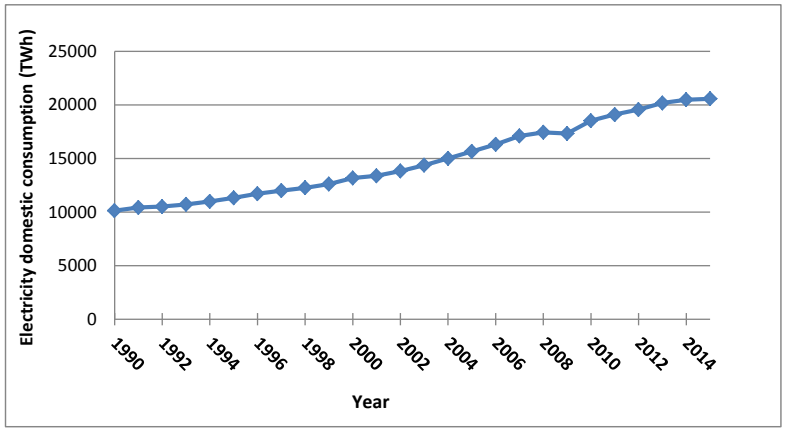


Figure 1.1: Worldwide energy consumption (y-axis) over years (x-axis) [ENE16]. The worldwide energy consumption rose at the rate was 56% from 2000-15.

of Energy and Nobel prize winner, Steven Chu placed this issue in the following context [CHU08].

'*A dual strategy is needed to solve the energy problem: (1) Maximise energy efficiency and decrease energy use. This part of the solution will remain the lowest hanging fruit for the next few decades; (2) Develop new sources of clean energy.*'

Consumer electronics are no different in contributing to high energy consumption. For example, according to Fraunhofer USA, consumer electronics devices in USA consumed 169 TWh of electricity in 2013, which amounts to 12% of residential electricity consumption [FRA14].

To avoid harmful effects of this trend, such as depletion of energy sources, higher emission of CO_2, and global warming, utilisation of *green computing* methods and practices must be observed at an individual level. By green computing, we refer to using energy-efficient and environmentally friendly electronic devices, refurbishing and recycling existing old electronic devices, and buying green electricity supplied from renewable energy sources.

1.2 Challenges: A Consumer Perspective

1.2.1 Longer System Lifetime

Modern embedded multimedia systems are equipped with ever increasing functionalities. If we consider the evolution of mobile phones, we can see that the a device whose sole purpose was to provide convenient communication, has become a true multimedia system. Apart from video streaming, current mobile phones are equipped with high quality cameras, and are able to browse the Internet, provide navigation and gaming interfaces etc. Cisco predicts that by 2020, 75% of the world's mobile data traffic will consist of multimedia content,

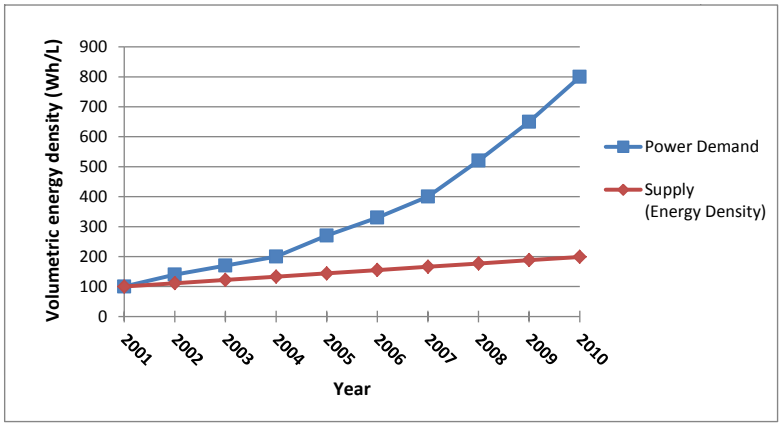


Figure 1.2: Battery energy density (y-axis) over years (x-axis) [ECO05], showing that the mobile device power consumption outgrows the amount of energy a battery can store, at a rate of more than three.

up from 55% in 2015 [CIS16]. Multimedia applications are considered as the most energy-hungry applications. Hence, a key challenge in modern embedded systems is the ever increasing energy consumption.

However, the battery energy densities of embedded multimedia systems have not grown at the same rate over the years [Cha07]. Figure 1.2 shows the growth of battery energy density versus power demand, according to a study by the Boston Consulting Group in 2005 [ECO05]. According to this study, the amount of energy that a battery can store (its *energy density*) is growing by 8% a year. Mobile-device power consumption, meanwhile, is growing at more than three times this rate. As a consequence, everyone owning a mobile phone is aware of the issue to monitor the battery charge and recharge it frequently. Not only mobile phones, every battery-powered system faces the same challenge. For example, Tesla's Model S electric car with a 60 kWh battery delivers 206 miles (334 km) [EPA13]. Therefore, for long trips, the driver has to continuously monitor the battery level, and get it charged at regular charging points.

Thus, *system lifetime* is a major challenge that consumers have to face all the time, i.e., the time one can use the battery before it is empty.

1.2.2 Robust Performance

Modern embedded multimedia systems are expected to perform robustly under strict resource constraints. A mobile phone capable of playing HD videos is a typical example. To process a video frame, the audio and video streams are split and processed separately. The video stream undergoes various picture enhancement steps to improve the video quality. Similarly, several audio improvement algorithms, e.g., echo cancellation, noise reduction etc., are performed on the audio stream. After the audio and video streams are processed separately, they are put in sync again and output on the screen and speakers. Hence, seamless

and robust performance is a key requirement for consumers.

Modern day mobile phones such as Google Pixel [GOO16] are able to support VR, which has even higher video quality and resolution than HD videos. Thus, VR requires more intensive processing than HD videos, which in turn requires more processing power, to provide the same seamless performance.

1.3 Challenges: An Industry Perspective

Gordon Moore predicted in 1965 that processing power for computers will double every two years [Moo65]. Over the past half-century, engineers more or less managed to maintain that predicted pace. As a result, software applications with increasing concurrency and complexity are continuously implemented on embedded multimedia systems. Mobile phones, discussed earlier, are a typical example of devices with increasing complexity. We see the same trend of increasing complexity also in other embedded multimedia systems such as VR-enabled gaming consoles, TVs, and cameras. Frits Vaandrager predicted this trend in 1998 by stating [Vaa98]:

'*In recent years there has been a dramatic growth of the number of embedded applications and of the size and complexity of the software used in these applications. For many products in the area of consumer electronics the amount of code is doubling every two years.*'

To cope with the ever-increasing complexity and deliver robust performance, modern-day embedded multimedia systems must possess sufficient *computational power*. At the same time, energy consumption must be kept to a minimum as most of these devices are battery powered (e.g., mobile phones, tablets, satellites, portable gaming consoles etc.). Thus, we cannot add processors more than a certain extent due to strict energy limitations. In addition to energy, these devices also have size and cost limitations which further restrict the number of processors.

To minimise energy consumption and prolong system lifetime, modern processors are being equipped with several energy management techniques, e.g., adapting the speed of the system to balance energy and performance implemented as *Dynamic Voltage and Frequency Scaling* [WWDS94], sleep modes implemented as *Dynamic Power Management* [BBDM00], and partitioning processors as *Voltage and Frequency Islands* [HM07]. This thesis deals with all of these energy management methods. Furthermore, the processors in a hardware platform can be classified as: (1) *homogeneous* where all processors are identical, so a (streaming) task can be mapped on any processor, or (2) *heterogeneous* where a (streaming) task cannot be mapped to any processor.

Moreover, in modern embedded multimedia systems, different components are interconnected, and hence influence each other. Thus, it is not easy to separate different design concerns such as computation, communication, power consumption, memory storage etc., and then try to integrate them together in a naive way. Other than rapidly evolving technology, the fierce market competition puts extra pressure on system designers to shorten time-to-market and reduce

development cost, while they are dealing with design complexities they have never seen before.

One solution to bridge this gap is by providing more design automation, e.g., by utilising computer-aided design (CAD) tools to assist with designing, synthesis, simulation, analysis, and testing process. By shifting the design tasks to computers, the minds of the system designers can be liberated to focus more on understanding the increasing complexity and how to handle it.

1.4 Problem Statement

Real-time embedded multimedia applications are often composed of several individual tasks. However, embedded multimedia systems have a limited number of processors to run these applications. To meet severe performance constraints such as functioning robustly while consuming as low as possible energy, efficient mapping of tasks to processors is necessary. Mapping an application onto a multiprocessor system involves three main operations: (1) assigning tasks to processors, (2) ordering tasks on each processor, and (3) specifying the time at which each task executes. These operations are collectively referred to as *scheduling* the application on the multiprocessor system.

In this thesis, we are interested in generating *time-* and *energy-optimal* schedules of streaming applications. From generated schedules, we can determine a trade-off between performance, energy consumption, and number of processors. This facilitates system designers to build robust systems with longer lifetime, and reduced development and manufacturing costs. The central research question addressed by this thesis is formulated as follows.

'*How to manage performance and energy of streaming applications running on a given number of (possibly heterogeneous) processors with respect to their hard real-time requirements.*'

1.5 Proposed Approach

For realising time- and energy-optimal scheduling of streaming applications, we need an approach with the following components, as shown in Figure 1.3.

- *Model of computation for streaming applications.* Firstly, we need a model of computation for streaming applications that captures all semantics of an application such as inter-task dependencies and their synchronisation properties. Furthermore, the model of computation must be able to express the timing behaviour of an application. This thesis considers *synchronous dataflow* (SDF) [LM87b] which is a popular formalism for modelling streaming applications.

- *Hardware platform model.* Secondly, a multiprocessor hardware platform model is required onto which the streaming applications' tasks can be mapped. The hardware platform model must also offer heterogeneous mapping capabilities, in case a task cannot be mapped to all processors

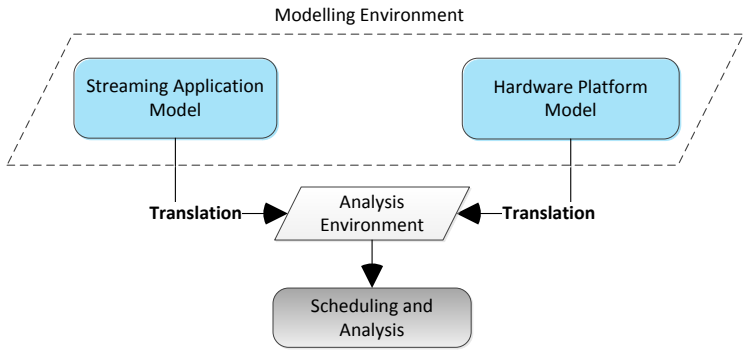


Figure 1.3: Overview of the approach proposed in this thesis. A streaming application is modelled, along with a hardware platform, using a modelling environment. Afterwards, these models are translated to an analysis environment which analyses performance and derives optimal schedules.

due to computation limitations. Moreover, the hardware platform model must be decorated with various timing- and energy-related aspects such as Dynamic Voltage and Frequency Scaling, Dynamic Power Management, and Voltage and Frequency Islands. To model hardware platforms, we introduce *platform application models* (PAMs) in this thesis.

- *Analysis environment.* Thirdly, we need an analysis environment for generating time- and energy-optimal schedules for an application. For this purpose, we model SDF and PAMs in *automata* and utilise *model checking* [CE81, QS82].

- *Modelling environment.* Lastly, we need an environment that allows efficient modelling of SDF, PAMs, and mappings of SDF tasks to hardware platforms. This is achieved with the help of *model-driven engineering* [VSB+13].

Figure 1.3 shows the flow of our approach and how the aforementioned components are related to each other. First, a streaming application captured by an SDF model of computation, and a hardware platform represented as a PAM, are modelled using model-driven engineering. Secondly, these models are translated to the model-checking domain, which is used to generate the optimal schedules and analyse the performance. In the following sections, we discuss the components in Figure 1.3 in more detail.

1.5.1 Synchronous Dataflow

In an SDF graph, actors communicate with each other by sending ordered streams of data-elements (termed *tokens*) over channels. When an actor fires, it consumes tokens from its input channels, performs computations on these

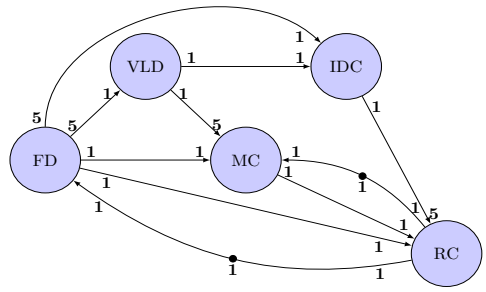
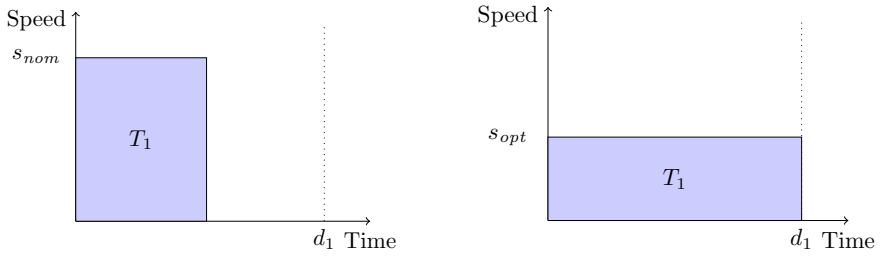


Figure 1.4: SDF graph of an MPEG-4 decoder. Each actor performs part of the MPEG-4 decoding such as, frame detection (FD), variable-length decoding (VLD), inverse discrete cosine transformation (IDC), motion compensation (MC), and reconstruction (RC).

tokens, and produces results as tokens on output channels. In an SDF graph, actors consume and produce a fixed amount of tokens when they fire. This type of model of computation makes it possible to analyse various features such as, a throughput [dKBS12, SBGC07, GGS$^+$06], latency [SB09, GSB$^+$07], and minimum buffer requirements [GBS05, HRG08, WBS07].

Example 1.1. MPEG-4 is a popular data compression method of audio and visual (AV) digital data. The SDF graph of an MPEG-4 decoder is shown in Figure 1.4 [TKW12]. Each of the five actors performs part of the MPEG-4 decoding. The MPEG-4 decoding starts in the actor FD (frame detector) which detects the type of the incoming frame. Different frame types require different number of *macroblocks*, which are processing units in image and video compression formats. The SDF graph in Figure 1.4 contains the number of macroblocks equal to five (shown by the number on the tail of the outgoing channels of FD to VLD and IDC). The actor VLD (variable-length decoder) decodes the variable number of bits, IDC (inverse discrete cosine transformation) applies the data decoding, and MC (motion compensation) predicts a frame in a video by accounting for motion of the camera and/or objects in the video. The complete frame is decoded when the video is reconstructed by the actor RC (reconstruction).

The actors are connected by the channels which correspond to (in principle unbounded) first-in first-out (FIFO) buffers. The actors communicate over the channels by exchanging tokens (unit of information that is communicated between the actors) represented by dots. For example, in Figure 1.4, the number of initial tokens in the channel from RC to FD represent how many frames the SDF graph can process concurrently. As we have one token in the channel from RC to FD in Figure 1.4, the SDF graph can process one frame at a time, and next frame can start only after the completion of the first frame. □



(a) Schedule without DVFS. The speed of the processor s_{nom} is too high, as a result the task T_1 finishes well ahead of its deadline d_1, leading to higher energy consumption.

(b) Schedule with DVFS. The speed of the processor is decreased to s_{opt}, which makes the task T_1 to finish exactly at the deadline d_1, leading to reduced energy consumption.

Figure 1.5: Schedules without and with DVFS

1.5.2 Hardware Platform Model

The SDF actors are mapped onto a hardware architecture termed *Platform Application Model* (PAM). The PAM consists of multiprocessors, to handle the concurrency of an SDF application. Moreover, the PAM is able to capture timing aspects of SDF actors, and energy related features such as power consumption of the processors. As mentioned earlier, the energy optimisation for modern processors has become one of the most critical, challenging and essential criteria. Therefore, the PAM is equipped with energy management techniques, namely DVFS and DPM. We explain these techniques in the following.

Dynamic Voltage and Frequency Scaling

The speed (operations per second) of a processor scales cubically to its power consumption [GHK14]. Dynamic Voltage and Frequency Scaling (DVFS) [WWDS94] is a technique that decreases the clock frequency (and the voltage) of a processor, leading to reduced speed and power consumption. In this way, power is consumed for a longer time, but the overall global energy consumption[1] is reduced. Other than processors, DVFS is also used in devices such as flash storage, hard drives, and network cards [LK10, SC01].

Example 1.2. Let us consider an application in Figure 1.5a having one task T_1 with a finishing deadline d_1. The x-axis shows the time, and the y-axis shows the speed. The amount of work for task T_1 is represented by the area of the task (time $\times$ speed). We assume that this task is running at the nominal speed s_{nom}. As a result, it finishes well before its deadline.

Figure 1.5b shows the result after deploying DVFS, where we can see that the speed of the same task reduces, and it finishes exactly at the deadline d_1. As there is a cubic relation between the speed and the power consumption (energy per time unit), the overall energy consumption is reduced also. □

[1] Energy is the integral of power over time.

Task	Amount of Work	Arrival Time	Deadline
T_1	5	0	19
T_2	5	0	19
$T3$	5	14	19

Table 1.6: Task characteristics of the example application

Dynamic Power Management

We have seen earlier that DVFS reduces the energy consumption of a device when it is *active*. Another novel energy management technique termed Dynamic Power Management (DPM) [BBDM00] reduces the energy consumption of a device when it is *idle*. DPM supports switching to a low power *sleep mode*, if the device is idle for a longer period of time. For example, consider a processor of a typical mobile phone, having three power states, i.e., ON, INACTIVE, and OFF. If the processor is in the ON state, LCD and backlight of the phone is turned on. If the phone remains idle for some time, the processor enters the INACTIVE state in which the backlight turns off, but the LCD stays turned on. If the phone stays idle for some more time, the LCD is turned off too (the OFF state).

A device may have multiple sleep states. The deeper the sleep model, the higher are the energy savings at the expense of transition latency[2]. Therefore, a device is put to the sleep mode only if the energy savings at the sleep mode outweighs the energy costs of transitioning to the sleep mode and back. The combination of DPM and DVFS guarantees optimal energy reduction.

Example 1.3. Let us consider an application having three tasks given in Table 1.6, adapted from [Ger14]. We assume that these tasks are mapped on a single processor, and that a task cannot be interrupted after it has been started, i.e., preemption is not allowed. The processor has a single sleep mode where it consumes no power. The power consumption of the processor is 1 W when it is idle. The power cost for transition to the sleep mode and back is 2 W and 1 W respectively. We ignore the active power consumption of this example, as it does not affect the mapping order of the tasks.

From Table 1.6, we can observe that task T_3 cannot be started before 14 time units. Therefore, tasks T_1 and T_2 must be finished before 14 time units in any order. Please note that whatever the ordering sequence and the starting time of tasks T_1 and T_2, the total idle time of the processor is always equal to 4 time units.

Figure 1.7 shows one possible schedule of this example. Here, the processor is idle between 5 and 7 time units, where the total idle power consumption is 2 W. The total power cost of transition to the sleep mode and back is 3 W, which is larger than the idle power consumption. Therefore, the processor stays idle, and

[2]Transition latency is the time required to switch to a sleep model and back.

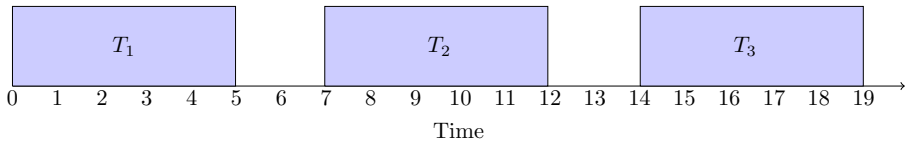


Figure 1.7: Suboptimal schedule (idle time energy = 4 W)

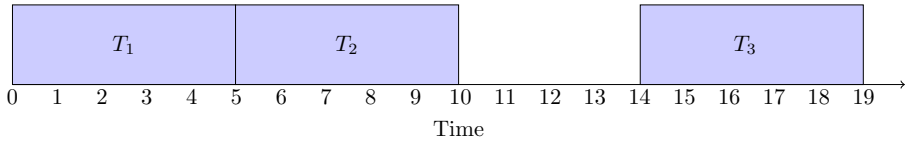


Figure 1.8: Optimal schedule (idle time energy = 3 W)

does not go to the sleep mode. The same happens when the processor is idle again between 12 and 14 time units. Hence, the total idle power consumption of this schedule is 4 W.

In an optimal schedule of this example, the processor stays idle for 4 time units continuously. In this way, the total power cost of the transition to the sleep mode outweighs the idle power consumption. Therefore, the processor moves to the sleep mode, which reduces the total idle power consumption to 3 W. Figure 1.8 shows one possible optimal schedule of this example, where the processor is idle for only once between 10 and 14 time units. □

Voltage and Frequency Islands

For multiprocessors, DVFS comes in two favours, viz., local and global DVFS [MSH+11]. While local DVFS changes the speed per processor, global DVFS makes this change for all processors. Local DVFS is the more energy-efficient of the two. However, local DVFS is expensive and complex to implement because it requires more than one clock domain. In contrast, global DVFS requires a simpler hardware design, but may lead to less reduction in power consumption [MSH+11].

To balance the energy efficiency and design complexity, the concept of voltage and frequency islands (VFIs) [HM07] is introduced. A VFI consists of a group of processors clustered together, with each VFI running at a common speed. The speed of the processors in the same VFI may differ from the processors in other VFIs.

Example 1.4. Figure 1.9 shows an example hardware platform model with 12 processors (adapted from [OMCM07]). The processors are partitioned into three VFIs shown by white, red, and green background colour. □

Scheduling Techniques

Now, that the methods for energy management in multiprocessors are presented, we will explain the different scheduling techniques in the following. Recall

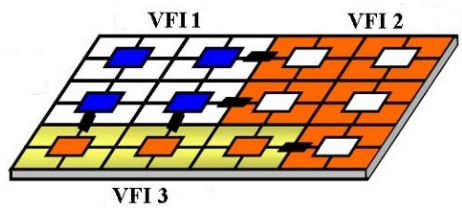


Figure 1.9: Hardware platform model having 12 processors partitioned into three VFIs shown by white, red, and green background colour.

that the scheduling problem consists of (1) assigning tasks to processors, (2) specifying the order in which the tasks fire on each processor, and (3) specifying the time at which the tasks fire. For generating energy-optimal schedules, the scheduling method needs to keep into account two additional parameters, i.e., (1) the optimal idle and running time of the processors (controlled through DPM), and (2) the optimal frequency to execute a certain task (throttled using DVFS).

Lee and Ha published a scheduling taxonomy based on performing tasks at either compile-time or run-time [LH89], as shown in Table 1.10. Each scheduling strategy is explained in the following.

Fully dynamic. The first strategy is *fully dynamic*, where all of the scheduling steps are performed at run-time. When all input operands for a task are available, the task is assigned to an idle processor, its order of firing is determined, and executed. The most common fully dynamic scheduling strategies are earliest deadline first (EDF), where the task having the earliest deadline is given priority, and rate-monotonic scheduling (RMS) where the task with the lowest amount of work is given priority.

Static-assignment. In *static-assignment* strategy, instead of assigning a task to a processor at run-time, this step is done at compile-time. Then, using a local run-time scheduler, tasks are assigned to a processor and executed.

Scheduling Strategy	**Assignment**	**Ordering**	**Timing**
Fully dynamic	Run	Run	Run
Static-assignment	Compile	Run	Run
Static-order	Compile	Compile	Run
Fully static	Compile	Compile	Compile

Table 1.10: Multiprocessor scheduling strategies

Static-order. In *static-order* strategy, the compiler assigns the processor to a task, as well as the order of firing. Afterwards, a local run-time scheduler executes the tasks when their input data is available.

Fully static. The last strategy is *fully static*; here the compiler determines the assignment, ordering, and the exact execution time of tasks. There exist different methods for devising fully static scheduling, e.g., round-robin and Time Division Multiplexing (TDM).

Fully static strategy is extensively used in scheduling of streaming applications because of its low implementation overhead [KCMH10]. Because of the same reason, this thesis also considers the fully static scheduling strategy.

1.5.3 Model Checking

For the analysis of timing and energy behaviour of a streaming application mapped on a hardware platform, we also need a suitable analysis environment. Nowadays, three performance analysis approaches are used for embedded applications, namely simulation, mathematical optimisation, and model checking. We explain each of the approaches in the following.

Classical Simulation. Simulation is the process of designing a model of a real system and conducting experiments with this model for the purpose either of understanding the behaviour of the system or of evaluating various strategies for the operation of the system [Sha75]. Simulation involves the following phases.

1. The first step is to generate an artificial history of the system.
2. The second step involves observation of the artificial history to derive inferences concerning the functional characteristics of the real system.

A specific simulation termed *Monte Carlo* allows probabilistic analysis of a system. This is done by repeated random sampling of the input variables based on their distributions to obtain the statistics of the output variables [Moo97, Fis96]. Monte Carlo method is guaranteed to terminate, but it is not guaranteed to give the correct result. Simulation explores only a *limited* set of possible execution of the system.

Mathematical Optimisation. Optimisation is an approach to find an optimal, or the absolutely most efficient, way to achieve an objective while simultaneously satisfying all constraints associated with achieving this objective [OPT, Sny05]. Typically, the objective is maximisation or minimisation of an analytical mathematical expression with a large amount of variables. A typical model in mathematical optimisation consists of the following four key ingredients [Kal04].

- data representing *constants* in a system such as production capacity of a manufacturing plant;
- variables representing *parameters* in a system such as production rate of a manufacturing plant;

Approach	Optimality	Schedule Generation	Ease of Modelling
Classical Simulation	−	−	+
Optimisation	+	+	−
Model Checking	+	+	+
Statistical Model Checking	+/−	−	+

Table 1.11: Comparison of different performance analysis approaches

- constraints representing restrictions of a system such as downtime of a manufacturing plant caused by breakdowns, material shortages etc., and
- the objective function representing the goal such as maximisation of utilisation rate of a manufacturing plant.

Model Checking. Model checking is a model-based verification technique that analyses the system behaviour in a mathematically precise and unambiguous way [CE81, QS82]. Using model checking, we can explore *all* possible states in a brute-force way. In this way, it can be shown that a system truly (dis)satisfies a property.

As mentioned earlier, we are interested in deriving optimal schedules. For this purpose, the analysis environment must have the following features.

- It must guarantee that the achieved results are optimal.
- It must be able to generate execution traces (schedules).
- It must be model-based, in order to fit in our model-driven engineering based framework.

Table 1.11 shows the comparison of these analysis approaches. Classical simulative methods are mostly model-based and generate traces. However, they cannot make sure that all interesting corner cases are covered even if we run simulations exhaustively, and thus optimality cannot be guaranteed. On the other hand, mathematical optimisation ensures optimality. However, mathematical optimisation is difficult to model as it requires quantification of all ingredients such as variables, constraints etc. in a mathematical form. Only model checking provides all of these features, and hence we consider it as an analysis environment in this thesis. In addition to model checking, we also consider statistical model checking. In contrast to classical simulation, statistical model checking combines simulations and statistical methods (such as sequential hypothesis testing) in order to decide with some degree of confidence whether the system satisfies the property or not.

In particular, we use model checking for performing *nondeterministic* scheduling, where the choices of the assignment, ordering, and the exact firing time of actors is determined nondeterministically by a scheduler at design-time in such a way that the generated schedules are time- or energy-optimal. In contrast, classical fully static strategies such as round-robin and TDM cannot guarantee optimality. For example, if we have an SDF graph where an actor rarely fires, we still have to assign a time slice to that actor in the round-robin scheduling strategy, which will affect the overall finishing time.

1.5.4 Model-Driven Engineering

To achieve optimal timing and energy management of a streaming application, the last component we need is an efficient modelling environment. This thesis considers the *Model-Driven Engineering* (MDE) [VSB+13] paradigm that treats models as first-class citizens. In MDE, the important concepts of the target domain are formally captured in a so-called *metamodel*. Separate metamodels for the domains of interest help to keep the design modular. All models are instances of a metamodel, or possibly an integrated set of metamodels. Moreover, a model can be transformed to another via *model transformations*, defined at the metamodel level. MDE also provides modularity, convenient extension mechanisms, and interoperability between different tools.

1.6 Thesis Structure

1.6.1 Thesis Overview

Figure 1.12 shows the structure of the thesis. The whole thesis is divided into the following three main parts.

- The first part *Background* contains Chapter 2 and 3. This part presents background material required to understand later chapters. Therefore, the readers are urged to study this part first.
- The second part *Analysis and Scheduling* contains Chapters 4 – 6. These chapters explain different scheduling and analysis techniques for SDF graphs mapped on multiprocessor hardware platforms. The chapters in this part can be read independently after reading the first part.
- The third part *Modelling and Validation* contains two chapters. Chapter 7 introduces model-driven engineering for dataflow applications. This chapter can be studied independently. Chapter 8 applies the methodology in Chapter 4 in the case study of face recognition system. Thus, the readers are advised to read Chapter 4 first before reading this chapter.

1.6.2 Contributions

This thesis makes several contributions to efficient modelling and optimal scheduling of streaming applications on a multiprocessor platform. Given an SDF graph, this thesis presents the following contributions.

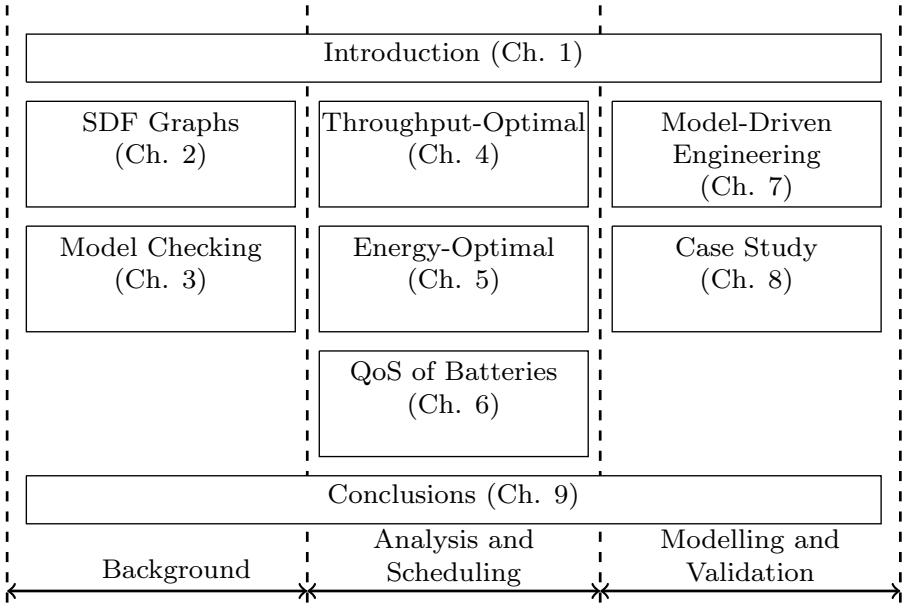


Figure 1.12: Overview of the structure of the thesis

- *Throughput optimisation*. A technique of deriving a schedule that fits on the given number of processors and maximises throughput is given (Chapter 4). This technique can also handle heterogeneous processor models, in which only specific processors can run a particular task due to their computational limitations. Moreover, using this technique, we can determine a trade-off between number of processors and throughput.

- *Energy optimisation*. An energy optimisation method that applies the combination of Dynamic Power Management (DPM) and Dynamic Voltage and Frequency Scaling (DVFS), and considers processors partitioned into Voltage and Frequency Islands (VFIs) is presented (Chapter 5). We further demonstrate that VFIs allow combining DPM and DVFS policy with any granularity.

- *Performance assessment*. An approach of assessing performance, and model checking of multiple batteries for different design alternatives is presented (Chapter 6). It is further shown that our approach allows better scalability than a state-of-the-art approach [JHBK09].

- *Efficient modelling using MDE*. A state-of-the-art model-driven engineering (MDE) framework is proposed (Chapter 7). In the framework, we present a reusable set of three coherent, extensible metamodels. Furthermore, we define and apply model transformations from the dataflow domain to the model-checking domain. Lastly, we demonstrate that our fully automated framework provides modularity, extensibility and interoperability between tools.

- *Practical Validation*. The technique of generating throughput-optimal schedules presented in Chapter 4 is validated (Chapter 8). For this purpose, an industrial case study termed "Face Recognition Application" provided by Recore Systems, Netherlands is considered.

1.6.3 Contents and Origins of the Chapters

The remainder of the thesis is organised in the following way. The origins of the chapters are also given where relevant.

- **Chapter 2** presents streaming applications and their characteristics. It also formally defines SDF graphs, and various notions associated with them. We also introduce the state-of-the-art tool for SDF analysis termed SDF^3.

- **Chapter 3** introduces the different model-checking formalisms considered in the thesis, and describes them with the help of examples.

- **Chapter 4** presents a technique of deriving throughput-optimal schedules of an SDF graph on a given number of processors, using model checking. This chapter is based on the paper "Resource-Constrained Optimal Scheduling of Synchronous Dataflow Graphs via Timed Automata", which was published at ACSD 2014 [AdGH$^+$14a].

- **Chapter 5** extends the work in Chapter 4 and presents a method of generating energy-optimal schedules of an SDF graph on a given number of processors, using model checking. This chapter is based on the paper "Green Computing: Power Optimisation of VFI-based Real-time Multiprocessor Dataflow Applications", which was published at DSD 2015 [AHSvdP15a].

- **Chapter 6** considers an intuitive battery model termed kinetic battery models (KiBaMs) [MG93] in the hardware platform model. In this way, the processors are dependent on the battery charge to run. Once the batteries are empty, the processors cannot run any more. Moreover, using statistical model checking, different performance aspects are determined. This chapter is based on the paper "Model Checking and Evaluating QoS of Batteries in MPSoC Dataflow Applications via Hybrid Automata", which was published at ACSD 2016 [AJSvdP16a].

- **Chapter 7** proposes a MDE-based approach for SDF graphs. This chapter is based on the paper "A Model-Driven Framework for Hardware-Software Co-design of Dataflow Applications", which was published at CyPhy 2016 [AYRS16a].

- **Chapter 8** performs the scheduling of an industrial case study of face recognition system mapped on a limited number of processors.

- **Chapter 9** concludes this thesis and provides future directions.

1.7 Conclusions

This chapter has laid the foundation for the rest of the thesis. In particular, the problem statement of the thesis and the proposed approach is explained. Over the past half century, the revolution in the hardware industry has changed the shape of the modern-day embedded systems. As a result, more and more applications and functionalities are being integrated in these systems. On the one hand, this has improved the standards in our daily life. On the other hand, this trend is causing negative effects on our environment due to increased emission of greenhouse gases. Furthermore, many of the embedded systems are battery-powered. This means that these systems must be recharged more frequently, which is leading to depletion of the world's energy sources. To cope with such situation, green computing has become an important, crucial, and key necessity of today's world.

This thesis contributes in overcoming this challenge by presenting an approach which allows efficient energy management of multiprocessor streaming applications, leading to energy-conscious systems. To realise this objective, the following choices are considered in this thesis.

- Streaming (software) applications modelled as SDF graphs.
- Homogeneous/Heterogeneous multiprocessor hardware platform to execute the streaming (software) tasks.
- Utilisation of MDE for efficient modelling of SDF graphs, hardware platform models, and mapping of SDF actors to processors.
- Model checking for performance analysis and generation of optimal schedules.

Part I

Background

CHAPTER 2

Dataflow Preliminaries

Algorithms for streaming applications can be naturally represented by block diagrams or flow charts in which computational blocks are interconnected by links that represent sequences of data values. Furthermore, the utilisation of visual programming in block diagrams or flow charts provides an intuitive specification mechanism for streaming applications. This thesis considers synchronous dataflow (SDF) [LM87b] to represent streaming applications

One of the reasons for the popularity of SDF models is their ability to capture parallelism in a streaming application. The imperative programming languages such as C and FORTRAN are based on *von Neumann* architecture, in which a small processor is attached to a big memory. Data items are present in the memory in their "cells" from where they are fetched one by one. Afterwards, these data items are sent to the central processing unit (CPU) in which the actual computation is performed, and then the results are returned one by one to their original cells. Thus, data in von Neumann architecture is static. This leads to a significant overhead of data-dependency constraints, resulting in challenges in compilation of such specifications onto the parallel hardware architectures. SDF models, on the other hand, impose minimal data-dependency constraints, enabling a compiler to detect parallelism. This also leads to efficient hardware synthesis, where it is important to specify and exploit concurrency.

Another reason for the popularity of SDF models is that they offer several analytical properties. The most important analytical property of SDF models is to effectively exploit parallelism in a streaming application by scheduling computations onto multiple processors at design-time. Given such a schedule computed at design-time, we can extract information from it towards optimising the final implementation.

Due to the success of SDF in industry, several commercial and research tools have been developed around SDF and closely related models. Commercial tools include Signal Processing Worksystem (SPW) from Cadence [PLN92, BL91], COSSAP [RPM92] and Cocentric System Studio [BV00] from Synopsys, ADS from Agilent, LabVIEW from National Instruments [AK98], and System Canvas from Angeles Design Systems [MCR01]. Tools based on the SDF formalism developed at different research laboratories and institutes include DESCARTES [RPM92], DIF [HKB05], GRAPE [LEAP95], the Graph Compiler [VPS90], NP-click [SPRK04], PeaCE [SOIH97], PGMT [Ste97], Ptolemy [BHLM94], StreamIt [TKA02], the Warp Compiler [Pri92], Lustre [HCRP91], Lucid [WL85], and SDF^3 [SGB06].

Chapter Outline. SDF graphs are formally defined in Section 2.1. Different semantics associated with SDF graphs are presented in Section 2.2 and Section 2.3 explains how to model storage capacities in SDF graphs. Section 2.4 describes the throughput analysis of SDF graphs. A state-of-the art tool implemented with various SDF graph analysis and techniques termed SDF^3 [SGB06] is explained in Section 2.5. A comparison of SDF and other dataflow models is given in Section 2.6. Section 2.7 explains different case studies modelled as SDF graphs. Finally, Section 2.8 presents a summary of the chapter.

2.1 Synchronous Dataflow Models

In typical streaming applications, there is a set of tasks to be executed in a certain order. An important part of these applications is a set of periodically executing tasks which consume and produce fixed amounts of data. An SDF graph is a directed, connected graph in which these tasks are represented by *actors*, data communicated is represented by *tokens*, and (FIFO) buffers used to transport tokens between actors are represented by *channels*. Each channel is connected to precisely one producer and precisely one consumer. The execution of an actor is known as an (*actor*) *firing* and the number of tokens consumed or produced onto a channel as a result of a firing is referred to as *consumption* and *production rates* respectively.

Example 2.1. Figure 2.1 shows an SDF graph with three actors u, v, w. Arrows between the actors depict the channels which hold tokens (dots). The numbers near the source and destination of each channel are the rates.

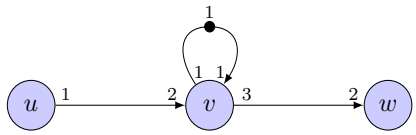


Figure 2.1: An example SDF graph (adapted from [dKBS12]).

Formally, the definition of an SDF graph is as follows.

Definition 2.2. *An* SDF graph *is a tuple* (A, D, Tok_0) *where:*

- A *is a finite set of actors,*
- D *is a finite set of dependency channels* $D \subseteq A^2 \times \mathbb{N}^2$*, and*
- $\mathsf{Tok}_0 : D \to \mathbb{N}$ *denotes initial tokens in each channel.*

A dependency channel $d = (a, b, p, q)$ *denotes a data dependency of actor* b *on actor* a*. The firing of actor* a *results in the production of* p *tokens on channel* d*. If the number of tokens on channel* d *is greater than* q*, actor* b *can execute, and as a result, it consumes* q *tokens from channel* d*.*

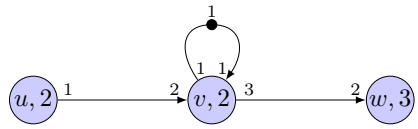


Figure 2.2: SDF graph in Figure 2.1 extended with time

Definition 2.3. *The sets of* input *In(a) and* output channels *Out(a) of an actor* $a \in A$ *are defined as*

$$In(a) = \{(a_0, a, p, q) \in D | a_0 \in A, p, q \in \mathbb{N}\}$$

$$Out(a) = \{(a, b, p, q) \in D | b \in A, p, q \in \mathbb{N}\}$$

Informally, if the number of tokens on every input channel d_i is greater than q_i, actor a_i fires and removes q_i tokens from every $(a_{0_i}, a_i, p_i, q_i) \in In(a)$. For example, actor v in Figure 2.1 consumes two tokens from channel u-v and one token from channel v-v, and produces three tokens on channel v-w and one token on channel v-v after finishing firing.

Definition 2.4. *The* consumption rate $CR(a, b, p, q)$ *and* production rate $PR(a, b, p, q)$ *of a channel* $(a, b, p, q) \in D$ *are defined as*

$$CR(a, b, p, q) = q$$

$$PR(a, b, p, q) = p$$

Synchronous Dataflow Graphs and Time. So far, the firings of the actors have been considered to be atomic. However, for analysing different system properties like throughput and energy optimisation, the notion of time is required to be associated with the firings of the actors in an SDF graph. In the following, a timed SDF graphs is defined by assigning a certain *execution time* to each actor [SB09].

Definition 2.5. *A* timed SDF graph *is a tuple* $G = (A, D, \mathsf{Tok}_0, \tau)$ *consisting of:*

- *an SDF graph* (A, D, Tok_0)*, and*
- *a function* $\tau : A \to \mathbb{N}_{\geq 1}$ *that assigns an execution time to each actor.*

Example 2.6. Figure 2.2 shows the SDF graph in Figure 2.1 extended with the execution times which are represented by a number inside the actor nodes. For example, actor v in Figure 2.2 takes 2 time units to finish its firing.

As we deal with the throughput and energy optimisation of the SDF graphs, we only consider timed SDF graphs in the rest of this thesis.

2.2 Semantics of SDF Graphs

The dynamic behaviour of an SDF graph G can be best understood if we define it in terms of a labelled transition system $\mathcal{LTS}(G)$. The LTS $\mathcal{LTS}(G)$ is defined by $(S, Lab, \rightarrow_G)$ where $S = (Tok, TuC)$ denotes the states, $Lab = \kappa$ denotes the labels, and $\rightarrow_G \subseteq S \times Lab \times S$ depicts the transitions. The ingredients of LTS $\mathcal{LTS}(G)$, i.e., states, labels, and transitions are defined in the following [GGS+06, SBGC07].

2.2.1 States

Definition 2.7. *The* state *of an SDF graph $G = (A, D, \mathsf{Tok}_0, \tau)$ is a pair (Tok, TuC) with the following components.*

- *$Tok : D \rightarrow \mathbb{N}$ associates with each channel the number of tokens it currently holds, and*
- *$TuC : A \rightarrow \mathbb{N}^{\mathbb{N}}$ records for each firing of actor $a \in A$ that occurred in the past, the remaining execution time. Thus, $TuC(a)(k)$ denotes that the remaining time of completion of different firings of $a \in A$ is exactly k time units. Here, TuC stands for "time until completion".*

The initial state of an SDF graph is defined as $(\mathsf{Tok}_0, \{(a, \emptyset) | a \in A\})$ where $\emptyset$ denotes an empty multiset.

Example 2.8. Suppose that the state vector of the SDF graph in Figure 2.2 is (Tok, TuC) where Tok corresponds to channels u-v, v-w, v-v respectively and TuC represents the multisets for actor u, v and w respectively. The initial state of the SDF graph in Figure 2.2 is $((0, 0, 1), (\emptyset, \emptyset, \emptyset))$. □

2.2.2 Auto-concurrency and Self-loops

By introducing the concept of multiset of numbers for actors, it is possible to have multiple simultaneous firings of same actor also known as *auto-concurrency*.

Example 2.9. Actor u in Figure 2.2 can fire multiple times simultaneously. □

Self-loops are used to restrict auto-concurrency of any actor with initial tokens on a self-loop equal to the desired degree of auto-concurrency.

Definition 2.10. *A channel $(a, b, p, q) \in D$ in an SDF graph is termed* self-loop *if $a = b$.*

Example 2.11. Channel v-v in Figure 2.2 is a self-loop. Since the number of tokens on channel v-v is one, actor v cannot fire more than one at a time. Hence, the degree of auto-concurrency is also one. If the number of tokens on channel v-v is increased to two and there are sufficient tokens on all other incoming channels, then actor v can fire twice simultaneously. Hence, the degree of auto-concurrency also increases to two. □

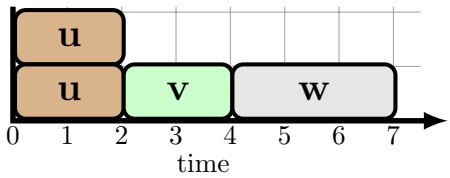


Figure 2.3: An example schedule of the SDF graph in Figure 2.2

2.2.3 Transitions

The transitions are of three forms, namely (1) the start transition labelled by *start* and actor name representing the start of actor firing, (2) the end transition labelled by *end* and actor name representing the end of actor firing, and (3) discrete clock ticks labelled by *tick* representing the progress of time. These transitions are defined in the following. To help understanding each transition, an example schedule of the SDF graph in Figure 2.2 is given in Figure 2.3. Each transition is explained with respect to this example schedule.

Definition 2.12. *A* transition *of an SDF graph* $G = (A, D, \mathsf{Tok}_0, \tau)$ *from state* (Tok_1, TuC_1) *to* (Tok_2, TuC_2) *is denoted as* $(Tok_1, TuC_1) \xrightarrow{\kappa} (Tok_2, TuC_2)$ *and label* κ *is defined as* $\kappa \in (A \times \{start, end\}) \cup \{tick\}$ *and corresponds to the type of transition.*

- *Label* $\kappa = (a, start)$ *denotes the starting of a firing by an actor* $a \in A$*. This transition may occur if for all* $a \in A$ *and* $d \in In(a)$*,* $Tok_1(d) \geq CR(d)$ *and results in* $Tok_2(d) = Tok_1(d) - CR(d)$ *and* $TuC_2(a) = TuC_1(a) \uplus \tau(a)$*.*

  Here $\uplus$ *represents multiset union; that is we remove* $CR(d)$ *tokens and attach* a*'s execution time* $\tau(a)$ *to* TuC_2 *for all* $a \in A$ *and* $d \in In(a)$*.*

  Example 2.13. The actor v in Figure 2.3 takes the transition $(v, start)$ at 2 time units. As a result, two tokens are subtracted from the channel $u - v$, and one token is subtracted from the channel $v - v$. □

- *Label* $\kappa = (a, end)$ *denotes the ending of a firing by an actor* $a \in A$*. This transition may happen if for all* $a \in A$ *and* $d \in Out(a)$*,* $0 \in TuC_1(a)$ *and results in* $Tok_1(d) + PR(d)$ *and* $TuC_2(a) = TuC_1(a) \backslash \{0\}$*.*

  Here $\backslash$ *represents multiset difference. This transition produces the specified number of tokens on the outgoing channel of* a *and removes from* TuC_1 *one occurrence of* a *with remaining executing time 0 for all* $a \in A$ *and* $d \in Out(a)$*.*

  Example 2.14. In Figure 2.3, the actor v takes the transition (v, end) at 4 time units. As a result, three tokens are produced on the channel $v - w$, and one token is produced on the channel $v - v$. □

- *Label* $\kappa = tick$ *denotes a clock tick transition. This transition is enabled if for all* $a \in A$ *and* $d \in D$*,* $0 \notin TuC_1(a)$ *and results in* $Tok_2(d) = Tok_1(d)$

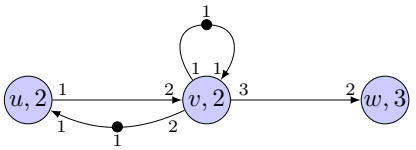


Figure 2.4: SDF graph in Figure 2.2 with an additional channel v-u.

and $TuC_2(a) = \{(a, TuC_1(a)) \ominus 1 | a \in A\}$ *where* $TuC_1(a) \ominus 1$ *denotes a multiset of elements of* $TuC_1(a)$ *decreased by one. This transition decreases by 1 the remaining execution time for all actor occurrences.*

Example 2.15. In our running example, there are two *tick* transitions between 2 and 4 time units because no end transition is enabled in that period. At 2 time units, the actor v starts firing. After two *tick* transitions, the remaining execution time of the actor v equals 0, and therefore end transitions is taken at 4 time units. □

In the following, the important concepts related to SDF graphs, i.e., execution, deadlock, and consistency are defined.

2.2.4 Execution

Definition 2.16. *An* execution *of an SDF graph* $G = (A, D, \mathsf{Tok}_0, \tau)$ *is a path in the LTS* $\mathcal{LTS}(G)$ *defined as a sequence of states and transitions* $\chi = s_0 \xrightarrow{\kappa_0} s_1 \xrightarrow{\kappa_1} \ldots$ *starting from initial state of SDF graph such that* $s_n \xrightarrow{\kappa_n} s_{n+1}$ *for all* $n \in \mathbb{N}$. *An execution is maximal if and only if it is finite with no transitions enabled in the final state, or if it is infinite.*

2.2.5 Deadlock

SDF graphs may end up in a deadlock due to inappropriate initial tokens in case of non-terminating programs.

Definition 2.17. *An SDF graph contains a* deadlock *if and only if it has a maximal execution of finite length [GGB+06].*

Example 2.18. Assume that in the SDF graph in Figure 2.2, we add a channel from actor v to u having a single initial token as shown in Figure 2.4. After a single firing of actor u, one token will be consumed from channel v-u and produced on channel u-v. Now actor u cannot fire any more as there is no token on channel v-u. Furthermore, actor v also requires two tokens on channel u-v to fire where there is only one token. Hence, the SDF graph cannot proceed and is in the deadlock state.

However, if we increase the number of initial tokens in the channel v-u from one to two, the SDF graph is deadlock free.

2.2.6 Consistency

SDF graphs may also suffer from unbounded accumulation of tokens in a certain channel due to inappropriate consumption and production rates. To avoid this effect, there is a property termed *consistency* which must hold [Lee91]. Consistency is defined in terms of *repetition vector*, which is a classical method for ensuring that there is no unbounded accumulation of tokens on any channel. For consistency to hold, the SDF graph must be connected. The repetition vector is defined as follows.

Definition 2.19. *A* repetition vector *of a connected SDF graph* $(A, D, \mathsf{Tok}_0, \tau)$ *is a function* $\gamma : A \to \mathbb{N}_0$ *such that for every channel* $(a, b, p, q) \in D$ *from* $a \in A$ *to* $b \in A$*, the following relation exists.*

$$p.\gamma(a) = q.\gamma(b)$$

Repetition vector γ *is termed non-trivial if and only if* $\gamma(a) > 0$ *for all* $a \in A$*. An SDF graph is consistent if it has a non-trivial repetition vector.*

The relation in Definition 2.19 can be written in the form of matrix-vector as:

$$\Gamma\gamma = \mathbf{0}, \tag{2.1}$$

Here Γ is termed as the *topology matrix* of an SDF graph and $\mathbf{0}$ is a null vector. The rows in Γ are indexed by the channels $(a, b, p, q) \in D$, and columns by the actors $a \in A$ in an SDF graph. For every channel $(a, b, p, q) \in D$ from $a \in A$ to $b \in A$ and actor $a' \in A$, the entries of the topology matrix are defined as:

$$\Gamma((a, b, p, q), a') = \begin{cases} p, & \text{if } a' = a \\ -q, & \text{if } a' = b \\ 0, & otherwise. \end{cases} \tag{2.2}$$

The formulation 2.1 assumes that the SDF graph does not contain any self-loops, channels whose source and sink actors are identical, such as channel v-v in Figure 2.4. In such a case, if we have a self-loop d', then source and target actor of d' are same which makes equation 2.2 self contradictory. In an SDF graph, a self-loop rules out the possibility of having a schedule if $p \neq q$; otherwise it does not have any effect on the existence of a schedule and is therefore not added to the topology matrix.

From equations 2.1 and 2.2, it follows that the repetition vector of an SDF graph with n actors numbered from 1 to n is a column vector of length n. If each actor a' is invoked a number of times equal to the $a'th$ entry of γ, then the number of tokens in each channel of the SDF graph remains unchanged. Thus, a repetition vector generates a finite length schedule, while avoiding unbounded accumulation of tokens on the channels (i.e., consistency). By repeating this finite length schedule indefinitely, one can generate infinite schedules of an SDF graph.

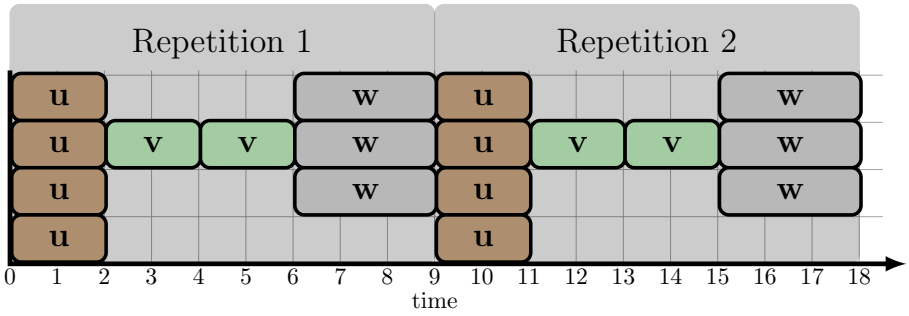


Figure 2.5: An example schedule of the SDF graph shown in Figure 2.2. The shaded regions show a finite length schedule in which all actors fire according to their repetition vector entries.

Example 2.20. Consider the example SDF graph in Figure 2.2, for which the topology matrix is given below.

$$\Gamma = \begin{pmatrix} 1 & -2 & 0 \\ 0 & 3 & -2 \end{pmatrix}$$

If we solve the relation $\Gamma\gamma = \mathbf{0}$ with Γ given above, the repetition vector γ results in $\gamma = \langle 4, 2, 3 \rangle$. Thus, the graph is consistent. An example schedule consisting of four firings of actor u, two firings of actor v, and three firings of actor w is shown in Figure 2.5. Here, we are repeating this schedule for two times. In the same fashion, we can generate an infinite schedule of this SDF graph. □

In the following, we discuss some concepts related to consistency of an SDF graph, taken from [LM87a]. Consider a tree-structured graph. This is a connected graph with no cycles, ignoring the direction of the channels.

Lemma 2.21. *A topology matrix Γ for a tree-structured SDF graph with n actors has rank $n - 1$.*

Proof. Proof is by induction. This lemma is clearly true for two-actor SDF graph. Assume that the lemma is also true for some n-actor SDF graph, with a topology matrix $\Gamma_n = n - 1$. Adding one node and one channel connecting the new actor to the graph will yield $n + 1$-actor SDF graph. A topology matrix for the new graph can be constructed from the old one by adding one column for the new actor and one for the new channel, as follows:

$$\Gamma_{n+1} = \left[\begin{array}{c|c} \Gamma_n & \mathbf{0} \\ \hline \rho^T & \end{array}\right] \tag{2.3}$$

where $\mathbf{0}$ is a column vector full of zeros, and ρ^T is a row vector corresponding to the new channel. The last entry in ρ^T must be nonzero since it indicates the number of tokens consumed or produced by the new actor on the new

channel. Hence, the last row is linearly independent from the other rows, so $rank(\Gamma_{n+1}) = rank(\Gamma_n) + 1$. Since any tree-structured SDF graph can be constructed by starting with a two-actor SDF graph and adding one more actor and channel repeatedly, $rank(\Gamma_n) = n - 1$ for any n. □

Lemma 2.22. *A topology matrix Γ for a connected SDF graph with n actors either has rank n or $n - 1$.*

Proof. First, it is obvious that the topology matrix Γ cannot exceed n, because Γ has only n columns. Now, we show that rank of topology matrix Γ cannot be less than $n - 1$. Consider any spanning tree of a connected SDF graph (a spanning tree is a tree that includes every actor in the graph). This spanning tree is a subgraph, i.e., it has all the actors of the original SDF graph but only a subset of the channels. Assume Γ_T to be the topology matrix for this subgraph. Following lemma 2.21, $rank(\Gamma_T) = n - 1$. Adding channels to the subgraph simply adds rows to the topology matrix. Adding rows to a matrix can simply increase the rank, if the rows are linearly independent of existing rows, but cannot decrease it. Hence, the rank of Γ cannot be less than $n - 1$. □

Lemma 2.23. *For a connected SDF graph with n actors and topology matrix Γ, $rank(\Gamma_n) = n-1$ is a necessary condition for the SDF graph to be consistent.*

Proof. If a nonzero repetition vector γ in $\Gamma\gamma = \mathbf{0}$ (equation 2.1) is required, it implies that $rank(\Gamma) < n$, where n is the dimension of γ. From lemma 2.22, $rank(\Gamma)$ is either n or $n - 1$, so it must be $n - 1$. □

If Γ has a rank $n-1$, we obtain the following facts by applying linear algebra [LM87a]:

Fact 2.24. *There exists a vector $\gamma \neq 0$ such that $\Gamma\gamma = 0$.*

Fact 2.25. *If $\Gamma\gamma = 0$ then $\Gamma(K\gamma) = 0$ for any constant K.*

Fact 2.24 concludes from lemma 2.23. Fact 2.25 explains that if a repetition vector γ of a connected SDF graph exists, i.e., the SDF graph is consistent, then the SDF graph is also consistent for any multiple of γ. In the remaining chapters, we always assume consistency.

2.3 Modelling Channel Capacities

An SDF graph typically only models an application. When mapping an application onto a hardware platform, the chosen platform imposes an extra set of constraints, that we need to take into account. Communication between actors in an SDF graph requires channel storage capacity. In case of an uniprocessor, we can consider to have a single memory that is shared between all channels. This provides us with the maximum number of tokens stored at any time during the execution of the SDF graph, by which we can determine the required channel capacities. In context of multiprocessor where memories are not always shared between all processors, we can use a separate memory for

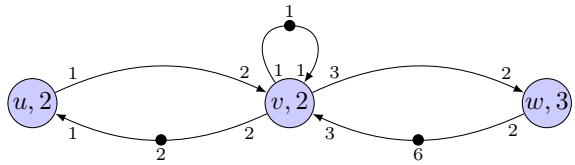


Figure 2.6: SDF graph shown in Figure 2.2 with channel capacities

each channel. Minimisation of the memory space for multiprocessors is studied in [BELP96, BML99, GBS05]. In this thesis, we assume that each channel has its own memory space which is not shared with other channels. We therefore define a channel capacity function, which yields the maximum number of tokens that can be stored on a channel.

Definition 2.26. *The* channel capacity *of an SDF graph G is a function $\varepsilon : D \to \mathbb{N} \cup \{\infty\}$ that assigns to each channel $d \in D$ the maximum number of tokens it can hold.*

The capacity of a channel $(a, b, p, q) \in D$ is modelled in an SDF graph by adding a channel $(b_\varepsilon, a_\varepsilon, q_\varepsilon, p_\varepsilon) \in D$ with $CR(a, b, p, q) = PR(b_\varepsilon, a_\varepsilon, q_\varepsilon, p_\varepsilon)$ and $PR(a, b, p, q) = CR(b_\varepsilon, a_\varepsilon, q_\varepsilon, p_\varepsilon)$ [Stu07]. The capacity of a channel $(a, b, p, q) \in D$ is denoted by the number of initial tokens on the channel $(b_\varepsilon, a_\varepsilon, q_\varepsilon, p_\varepsilon) \in D$.

Example 2.27. The SDF graph shown in Figure 2.2 after adding the channel capacities is shown in Figure 2.6. The channel capacities are $\varepsilon(u, v, p, q) = 2$ and $\varepsilon(v, w, p, q) = 6$. Now, the number of tokens on channels (u, v, p, q) and (v, u, p, q) cannot exceed 2. Same is the case for channels (v, w, p, q) and (w, v, p, q) where the number of tokens never exceeds 6. □

2.4 Throughput Analysis of SDF Graphs

This section explains the throughput analysis of SDF graphs after introducing necessary terms. If we suppose that an SDF graph $(A, D, \mathsf{Tok}_0, \tau)$ has a repetition vector γ, then we define iteration as follows.

Definition 2.28. *An* iteration *is a set of actor firings such that for each $a \in A$, the set contains $\gamma(a)$ firings of a.*

Definition 2.29. *The* throughput *for an execution χ of an SDF graph G denoted as $TP(G, \chi)$ is the average number of graph iterations that are executed per time unit in χ. That is,*

$$TP(G, \chi) = \lim_{t \to \infty} \frac{iter(\chi, t)}{t}$$

where $iter(\chi, t)$ is the number of iterations that are finished up to time t of the execution χ.

2.4.1 Self-timed Execution

The maximal throughput of an SDF graph G is determined from a specific type of execution known as a *self-timed execution* [GGS+06] in which every actor fires *as soon as* it is enabled. This subsection is mostly based on the work in [GGS+06].

Definition 2.30. *An* execution $\chi = s_0 \xrightarrow{\kappa_0} s_1 \xrightarrow{\kappa_1} \ldots$ *of an SDF graph* $G = (A, D, \mathsf{Tok}_0, \tau)$ *in the LTS* $\mathcal{LTS}(G)$ *is* self-timed *if and only if for all states* $s \in \chi$ *it holds that,*

> *clock transitions occur when* $Tok_1(d) \not\geq CR(d)$ *for all* $a \in A$ *and* $d \in In(a)$*, i.e., when no start transitions are enabled.*

In the self-timed execution of an SDF graph, between two clock transitions, there can be some interleaving of simultaneously enabled start and/or end transitions. However, as the order in which these transitions occur is completely independent of each other, the final state before each clock transition, and hence also the state after each clock transition, is always the same. Self-timed SDF graph behaviour is therefore *deterministic* in the sense that all states immediately before and after clock transitions are completely determined and independent of the selected execution.

Theorem 2.31. [GGS+06] *For every consistent and strongly connected SDF graph, the state-space of a self-timed execution* χ_{st} *is of the form* $\chi_{st} = \chi_{pre} \cdot \chi_{per}^{\infty}$ *with a finite sequence of states* χ_{pre} *(transient phase) followed by a periodic sequence* χ_{per}^{∞} *repeated infinitely (periodic phase).*

Theorem 2.31 states that the state-space of an SDF graph consists of a particular shape. That is, it contains a *transient phase* followed by a *periodic phase*. The reason is that in a consistent SDF graph, every actor is dependent on tokens from other actors to fire. This ensures that the number of tokens accumulated on any channel is bounded. This leads to the fact that the auto-concurrency of an actor is bounded as only an finite number of copies of an actor can be firing at the same time. If the number of tokens accumulated on any channel and auto-concurrency of each actor is bounded, the number of states of an SDF graph in a self-timed execution is finite. This guarantees that a certain state that was visited before is revisited implying the fact that execution is then in the periodic phase. As explained earlier, each actor fires according to the repetition vector in the periodic phase. For each actor $a \in A$ in the SDF graph, we define its corresponding entry in the repetition vector γ as $\gamma(a)$.

Example 2.32. The self-timed execution χ_{st} of the SDF graph in Figure 2.6 is explained in Figure 2.7. It is worth noting that after two simultaneous firings of actor u, an iteration is completed every 9 time units and hence the throughput is $TP(G, \chi_{st}) = \frac{1}{9}$. Note that in each iteration, the actor u fires four times, v two times, and w three times which is exactly the repetition vector as proven in Theorem 2.31.

Similarly, the self-timed execution in terms of the state vector (Tok, TuC) of the same SDF graph is shown in Figure 2.8 where Tok corresponds to the

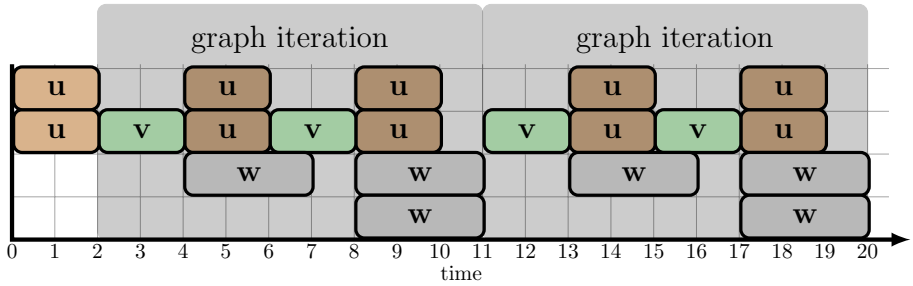


Figure 2.7: Self-timed execution of the SDF graph shown in Figure 2.6. After the transient phase (initial two firings of u), the SDF graph enters the periodic phase shown by the shaded regions. As the SDF graph takes 9 time units to finish one iteration in the periodic phase, the throughput is $\frac{1}{9}$.

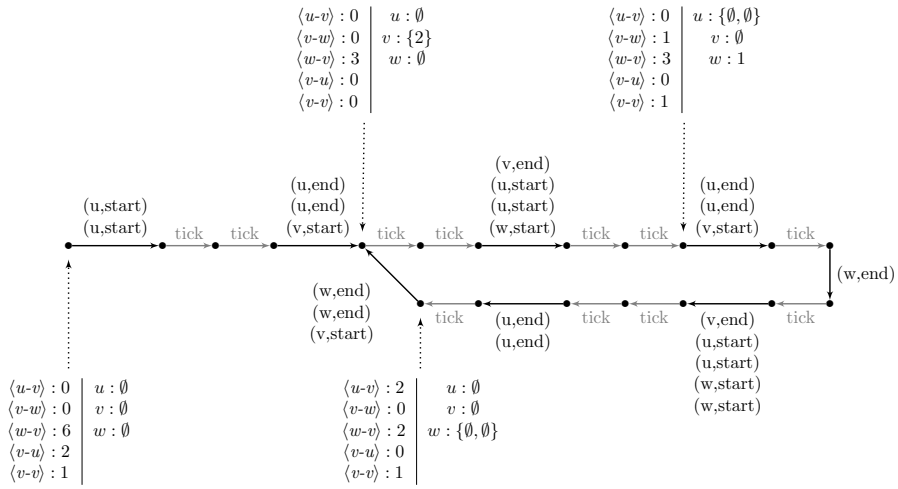


Figure 2.8: Self-timed execution of our running example in terms of state vector $(Tok,\ TuC)$ consisting of a transient and periodic phase.

channels *u-v, v-w, w-v, v-u,* and *v-v* respectively. Similarly, TuC corresponds to the multisets for actor u, v, and w respectively. We can also see that the state-space of the SDF graph consists of a transient and periodic phase. In the transient phase, the actor u fires twice, and in the periodic phase each actor fires according to its entry in the repetition vector. The periodic phase has a duration of 9 time units consisting of precisely one iteration. □

Example 2.33. Figure 2.9 shows another SDF graph having two actors y and z, with the repetition vector $\gamma = \langle 1, 1\rangle$. The self-timed execution χ_{st} of this SDF graph is shown in Figure 2.10. In this example, note that the periodic phase contains three graph iterations. Moreover, in each iteration, both actors y and z

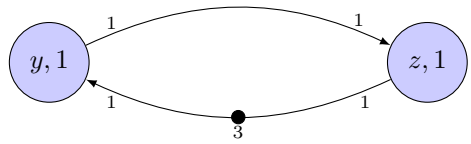


Figure 2.9: An example SDF graph

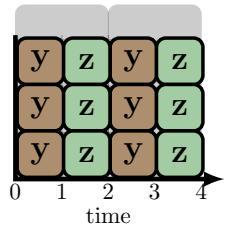


Figure 2.10: Self-timed execution of the SDF graph shown in Figure 2.9. The shaded regions show the periodic phase containing three graph iterations in which all actors fire according to their repetition vector entries.

fire once which is exactly the repetition vector. As the periodic phase takes 2 time units to finish, the throughput is $TP(G, \chi_{st}) = \frac{3}{2}$. □

The tool SDF^3 considers self-timed execution to calculate the throughput of SDF graphs. The detailed explanation of SDF^3 is given in the next section.

2.5 SDF^3: Synchronous Dataflow Analysis Tool

This section discusses a state-of-the-art tool for the analysis of SDF graphs termed **SDF For Free** (SDF^3) [SGB06]. Besides analysis, SDF^3 also supports visualisation of SDF graphs and generation of random SDF graphs. This thesis extensively utilises SDF^3 for modelling SDF graphs. Afterwards, these SDF graphs are automatically translated to the model-checking formalisms for scheduling using model-driven engineering. More details are given in Chapter 7. It is worth mentioning that SDF^3 assigns consumption and production rates to actors using the notion of *ports*. In contrast to SDF^3, we assign consumption and production rates to channels, thus simplifying the definition of SDF graphs.

For an SDF graph, SDF^3 supports computation of (self-timed) throughput and repetition vector. Furthermore, SDF^3 also has a support to check if an SDF graph is consistent, connected, or deadlock free. SDF^3 also offers algorithms to model required auto-concurrency of actors using self-loops, and model channel capacities using the technique explained in Section 2.3. A function to visualise SDF graphs through the popular graph visualisation tool termed dotty [GN00] is also integrated in SDF^3.

SDF^3 uses an XML-based format for designing SDF graphs. The layout of XML format used in SDF^3 is given in the next subsection.

2.5.1 Layout of SDF^3 XML

The main layout of XML specification for an example SDF graph in SDF^3 is given in Listing 2.1. Each element is explained in the following, taken from [SDF].

Listing 2.1: An example SDF graph modelled in XML format of SDF^3

```
1  <sdf3 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       version="1.0" type="sdf"
2        xsi:noNamespaceSchemaLocation="http://www.es.ele.tue.nl/
             sdf3/xsd/sdf3-sdf.xsd">
3  <applicationGraph name="app">
4    <sdf name="example" type="Example">
5      <actor name="a1" type="A">
6        <port name="p1" type="out" rate="1"/>
7      </actor>
8      <actor name="a2" type="A">
9        <port name="p1" type="in" rate="1"/>
10       <port name="p2" type="out" rate="1"/>
11     </actor>
12     ...
13     <channel name="d1" srcActor="a1" srcPort="p1" dstActor="a2"
           dstPort="p1" initialTokens="1"/>
14     ...
15   </sdf>
16   <sdfProperties>
17     <actorProperties actor="a1">
18       <processor type="p1" default="true">
19         <executionTime time="1"/>
20       </processor>
21     </actorProperties>
22   </sdfProperties>
23 </applicationGraph>
24 </sdf3>
```

sdf3. Line 1 assigns the value sdf to the *type* attribute of the *sdf3* element specifying that the XML file contains the specification of an SDF graph. The version of the XML syntax is specified in the (required) *version* attribute.

applicationGraph. The actual specification of an SDF graph is given in the *applicationGraph* element on line 3. This element occurs at most once in the XML file. The optional *name* attribute for the *applicationGraph* element allows specifying a name for the SDF graph, where the default value is no name.

sdf. The structure of the SDF graph is contained inside the *sdf* element on line 4. The *sdf* element has two required attributes. The first attribute is *name*, which is used to specify the name of the SDF graph. The second attribute is *type*, which is used to specify the type of the SDF graph.

actor. Lines 5-7 specify an actor of an SDF graph. The *actor* element is given on line 5. Similar to an SDF graph, an actor has a *name* and *type* attribute.

port. In SDF^3, an actor contains one or more ports which denote the consumption and production rates. Furthermore, actors connect to channels using these ports. Line 6 specifies a *port* element. The *port* element has three required attributes. The first attribute termed *name* specifies a logical name for the port. This name is used when connecting the port to a channel. The second attribute termed *type* can have the value *in* or *out*. The *type* attribute specifies if an actor consumes or produces tokens via that port. When a port is of type *in*, an actor consumes tokens from the channel to which the port is connected, during the firing. When a port is of type *out*, an actor produces tokens on the channel to which the port is connected, during the firing.

channel. Line 13 specifies a channel of an SDF graph with a *channel* element. A *channel* element has five required attributes. The first attribute termed *name* is used to specify the name of the channel. The remaining attributes are explained as follows.

- *srcActor*. The *srcActor* attribute represents the source actor of the channel.
- *srcPort*. The port on which the source actor is connected to the channel is denoted by the *srcPort* attribute.
- *dstActor*. The *dstActor* attribute represents the sink actor of the channel.
- *dstPort*. The port on which the sink actor is connected to the channel is denoted by the *dstPort* attribute.

The *channel* element contains an optional attribute termed *initialTokens* that represents the number of initial tokens present in that channel. When this element is absent, it is assumed that the channel contains zero initial tokens.

sdfProperties. While the structure of the SDF graph is contained inside the *sdf* element, the properties of the actors, channels and the graph are contained inside the *sdfProperties* element on line 16. The *sdfProperties* element may contain zero or more *actorProperties* used to specify the properties of the actors (e.g., execution time).

actorProperties. The *actorProperties* element on line 17 requires the *actor* attribute. The value of this attribute must contain the name of the actor to which the properties specified inside this element apply.

In SDF^3, the execution time of an actor is not attached directly to an actor. Rather, an actor is associated to a processor to which the execution time of the actor is associated. Thus, the *actorProperties* element must contain one or more *processor* elements.

processor. The *processor* element mentioned on line 18 requires the *type* attribute. The value of this attribute specifies the processor type for which the properties contained inside the element are valued. The *processor* element may also have a *default* attribute. This attribute can either have the value `true` or

Algorithm	Description
consistency	Checks consistency of the graph
deadlock	Checks absence of deadlock of the graph
connected_graph	Checks whether the graph is connected
repetition_vector	Computes the repetition vector of the graph
statistics	Computes some statistics (e.g., actor and channel count) of the graph
throughput	Computes the throughput of the graph
buffersize	Computes the throughput/storage-space trade-off point of the graph

Table 2.11: Analysis algorithms of SDF^3

false. When the *default* attribute is not given, its default value is false. The *default* attribute is used in case that more then one *processor* element is contained inside a *actorProperties* element. Then the value of the *default* attribute for one of the *processor* elements is set to true. The analysis algorithms of SDF^3 then only considers the *processor* element whose *default* attribute is set to true.

executionTime. The *processor* element must contain a *executionTime* element as mentioned on line 19. This element has one required attribute termed *time*. The value of this attribute specifies the (worst-case) execution time (in time-units) of the specified actor. SDF^3 uses this attribute to calculate the self-timed throughput of an SDF graph.

2.5.2 Analysis Algorithms of SDF^3

SDF^3 contains several SDF graph analysis algorithms. The command line options of SDF^3 are as follows:

sdf3analysis−sdf −−**graph** < `file` > −−**algo** < `algorithm` >

The mandatory argument −−graph specifies the link to the file containing the SDF graph. The another mandatory argument −−algo specifies the analysis algorithm to be executed. Some of the relevant algorithms are given in Table 2.11.

2.6 Comparison of Dataflow Models

There exists several variations of models of computation (MoCs) for dataflow, modelling different aspects of dataflow applications. This section, based on the

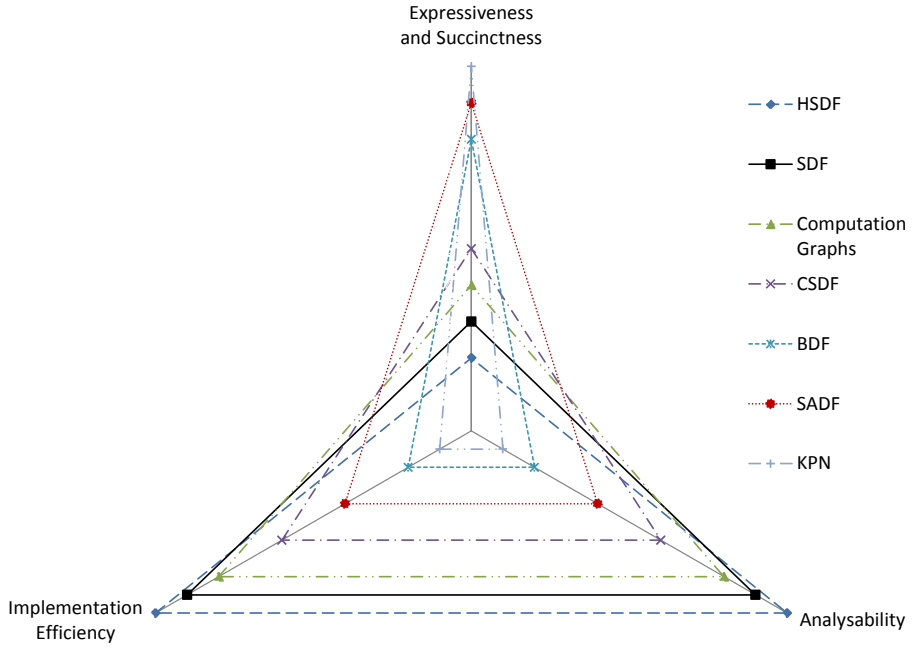


Figure 2.12: Comparison of dataflow models of computation (taken from [Stu07])

work in [Stu07, dG16], compares these MoCs.

A MoC must have the following three features for designing and analysing a system.

- *Expressiveness and Succinctness.* The first step in a system design and analysis is to model it. Expressiveness refers to which type of system properties, a model can capture, and succinctness refers to the compactness of these models.

- *Analysability.* After modelling a system, the second step is to analyse it. The analysability of a MoC is determined from how many algorithms are available to analyse different system properties. Moreover, analysability also includes the run time of these algorithms on SDF graphs with the given number of actors, independent of the considered hardware platform.

- *Implementation efficiency.* After a system is modelled in a certain MoC, and analysed, the last step is to generate schedules. Implementation efficiency refers to the complexity of the scheduling problem, and the (code) size of the generated schedules.

Different MoCs for dataflow are discussed with respect to these features in the following. The MoCs are divided into two types, (1) ones dating to SDF graphs, and (2) ones extending SDF graphs. The comparison of different MoCs

MoC	Distinctive Feature	Tool Support
Computation Graphs	Earliest notion of dataflow in the context of parallel computing	Yes
HSDF Graphs	Simplistic MoC leading to faster analysis	Yes
KPNs	Models data-dependency	Yes
SDF Graphs	Models multi-rates	Yes
CSDF Graphs	Models dynamic behaviour	Yes
BDF Graphs	Models data-dependency	Yes
SADF Graphs	Models data-dependency and dynamic behaviour	Yes

Table 2.13: Distinctive features of dataflow models of computation

is given in Figure 2.12 [Stu07]. Table 2.13 summarises the distinctive features of these MoCs.

2.6.1 Models of Computation dating to SDF Graphs

Computation Graphs. Computation graphs [KM66] are introduced by Karp and Miller in 1966. Similar to SDF graphs, actors in computation graphs consume and produce tokens on channels, and each channel contains initial tokens. Therefore, each channel in a computation graph has a consumption and production rate, and initial tokens. However, in contrast to SDF graphs, channels in computation graphs have one additional parameter, i.e., a threshold value indicating the minimum number of tokens that must be present on the channel before tokens may be read from it.

The distinction made in computation graphs between the minimum number of tokens needed for an actor to fire and the actual number of tokens consumed by a firing makes computation graphs more succinct. However, this extra condition also makes analysis algorithms more complex.

(Homogeneous) Synchronous Dataflow. Homogeneous Synchronous Dataflow (HSDF) graphs correspond to a subclass of *Petri Nets* [Pet62] termed *marked graphs*. HSDF graph are studied by Raymond Reiter in 1968, as a special case of computation graphs [Rei68]. In an HSDF graph, an actor consumes a *single* token from each of its incoming channel, on the start of its firing. An actor finishes its firing by producing a *single* token on each of its outgoing channel. HSDF graphs can be converted into SDF graphs and vice-versa [KCMH10].

Kahn Process Networks. In 1974, Gilles Kahn wrote an influential paper introducing a MoC for distributed systems, termed Kahn Process Networks (KPNs) [Kah74]. Gilles Kahn termed these networks "a simple language for parallel programming". In KPNs, processes communicate with each other by sending data over unbounded first-in first-out channels. A process can write to a channel

whenever it wants. But, to read an empty channel, the process blocks and waits until the data is available. The key idea of KPNs is that the networks of sequential processes ensure *functional determinacy*, which means that, when fed with the same input sequence, the computation yields the same output sequence.

KPNs are sufficiently expressive to capture all possible properties of a dataflow application. But, to analyse different system properties such as throughput and buffer requirements, one need to know all possible inputs, and every input may require a different schedule. Therefore, KPNs have lower analysability and implementation efficiency.

Synchronous Dataflow. Synchronous Dataflow (SDF) graphs [LM87b] introduced in 1987, deal with multi-rate data dependencies. In terms of Petri Nets, SDF graphs correspond to *weighted marked graphs* [TCWCS92]. In an SDF graph, actors consume and produce at a fixed but different rates. In comparison to HSDF graphs, SDF graphs can have multi-rates.

As regular SDF graphs deal with multi-rates, they are more expressive and succinct than HSDF graphs. But, this also means that the complexity of the analysis algorithms for SDF graphs is higher (non-polynomial). In contrast, the complexity of the analysis algorithms for HSDF graphs is polynomial. Thus, HSDF graphs are more analysable than SDF graphs. Dealing with multi-rates also leads to having complex scheduling problems in case of SDF graphs. Therefore, the implementation efficiency of SDF graphs is lower than HSDF graphs where we only have single rates.

2.6.2 Models of Computation extending SDF Graphs

Cyclo-static Dataflow. SDF graphs represent static streaming applications, i.e., the consumption and production rates per firing of an actor are constant. Greet Bilson et. al introduced a variant of SDF graphs termed cyclo-static dataflow (CSDF) graphs in 1996 [BELP95, BELP96], which overcomes this limitation. In a CSDF graph, actor cycles through finite number of periodically varying phases. In each phase, the execution time of actors may change. Moreover, the production and consumption rates of channels may also change in each phase.

The fact that CSDF graphs are more dynamic than SDF graphs, makes CSDF graphs more expressive. But, this also means that the analysis algorithms are slower than the algorithms for SDF graphs. Moreover, the implementation efficiency of CSDF graphs would also be lower. Conversion of CSDF graphs into SDF graphs is studied in [PPL95].

Boolean Dataflow. Both SDF and CSDF graphs cannot capture data-dependent behaviour. This makes it impossible for an actor to choose between two inputs depending on the value of a token on a third input. The Boolean Dataflow (BDF) graph [Lee91] introduced by Edward Lee in 1991 fills this gap. BDF extends SDF with two additional actors termed *switch* and *select*. The switch actor reads the data from its input, and copies to one of its output based on the value of a boolean *control* token. The role of the select actor is reversed, i.e., it reads the data from one of its input based on the value of a boolean control token, and copies

to its outputs. Having an ability to model data-dependent behaviour makes BDF graph more expressive than SDF graphs. But this also makes it difficult to generate schedules at design-time. Therefore, run-time scheduling must be performed, which in turns reduces the implementation efficiency. Furthermore, run-time scheduling also leads to the fact that BDF graphs are less analysable than SDF graphs at design-time.

Scenario-Aware Dataflow. Scenario-Aware Dataflow (SADF) [TGB+06] is a recently introduced MoC, which extends SDF graphs with a concept of *scenarios*. Each individual scenario is modelled by an SDF graph. The scenarios represent dissimilar modes of operation originating, for example, from different parameter settings, sample rate conversion factors, or the signal processing operations to perform. Thus, as compared to SDF graphs, SADF graphs allow modelling of dynamic behaviour. Furthermore, each scenario in an SADF graph has data-dependent consumption and production rates controlled with a control actor. Hence, in contrast to SDF and BDF graphs, SADF graphs allow varying rates, which makes them more succinct than SDF and BDF graphs.

Because it is possible to change scenarios inside an iteration of an SADF graph, the general SADF MoC requires run-time scheduling. This makes its implementation not very efficient. Moreover, the execution time of the analysis algorithms of SADF graphs is longer compared to similar analysis algorithms on equally sized CSDF or SDF graphs [Stu07]. The reason is that SADF graphs can model more behaviours than SDF graphs and all behaviours need to be analysed. SDF^3 offers support to convert SADF graphs to equivalent SDF graphs.

This thesis focuses on generating optimal schedules for streaming applications. The MoC representing the streaming applications must offer novel and efficient analysis algorithms. Moreover, the MoC must have as simple as possible implementation, so that the generated schedules can be implemented on a hardware platform easily. If we look at Figure 2.12, we see that these requirements are fulfilled by HSDF, SDF, and CSDF graphs. The HSDF MoC is not chosen because models of realistic case studies can be very large. The CSDF MoC is not selected as its implementation efficiency is lower than SDF graphs. Therefore, we consider SDF to represent streaming applications in this thesis. Furthermore, SDF graphs lie in between HSDF and CSDF graphs in terms of generalisation, i.e., CSDF graphs generalise SDF graphs, and SDF graphs generalise HSDF graphs [dG16]. Therefore, the scheduling techniques discussed in thesis can be extended to CSDF and HSDF graphs.

2.7 Case Studies

2.7.1 MPEG-4 Decoder

MPEG-4 already introduced in Chapter 1, is a method of defining compression of audio and visual digital data. The processing unit in video compression is termed *macroblock*. A macroblock typically consists of 16×16 array of pixels. The two major picture types used in the different video algorithms are I and P. An I-frame is an "Intra-coded picture", representing a conventional static image

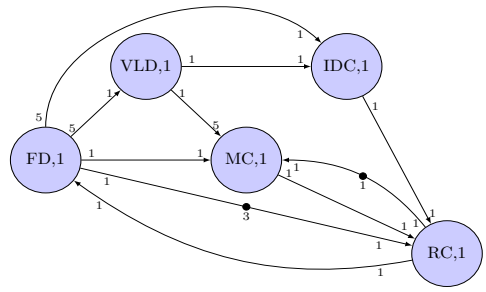


Figure 2.14: SDF graph of an MPEG-4 decoder [TKW12]

file. On the other hand, an P-frame is an "Predicted picture", and it carries only the changes in the image from the previous frame.

The SDF graph of an MPEG-4 decoder is shown in Figure 2.14 [TKW12]. Each of the five actors performs part of the MPEG-4 decoding, and are explained as follows.

- The MPEG-4 decoding starts in the actor *FD* (frame detector) which detects the type of the incoming frame. Different frame types require a different number of macroblocks. The SDF graph in Figure 2.14 contains the number of macroblocks equal to five (shown by the number on the tail of the outgoing channels of *FD* to *VLD* and *IDC*).

- The actor *VLD* (variable-length decoder) decodes the variable number of bits.

- The actor *IDC* (inverse discrete cosine transformation) applies the data decoding.

- The actor *MC* (motion compensation) predicts a frame in a video by accounting for motion of the camera and/or objects in the video.

- The complete frame is decoded when the video is reconstructed by the actor *RC* (reconstruction).

2.7.2 MP3 Decoder

MP3 decoding is a frame based algorithm that transforms a compressed stream of data into pulse-code modulation (PCM) data. The SDF graph of an MP3 decoder is shown in Figure 2.15 [DSB+11]. Each actor is explained as follows.

- MP3 decoding starts in the actor *Huffman*, where a specific variable-length decoding termed *Huffman* is performed.

- The next step is requantisation. This is done by the actors *Req0* and *Req1*. In this step, continuous input signals are converted to discrete ones.

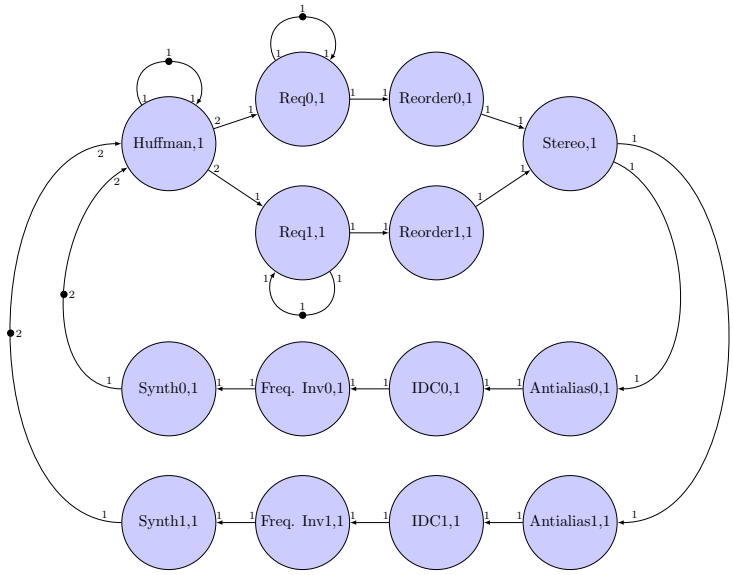


Figure 2.15: MP3 decoder [DSB+11]

- The output of the actors *Req0* and *Req1* is not always ordered in the same way. The actors *Reorder0* and *Reorder1* reorder the output generated in the requantisation step.
- MP3 has two different channel modes. The first mode termed *mono* means that the audio has a single channel. The other model termed *stereo* deals with the audio having two channels. In our case, we consider stereo. The actor *Stereo* combines the both audio channels.
- The actors *Antialias0* and *Antialias1* reduce a specific distortion in the audio channels termed *aliasing*.
- The actors *IDC0* and *IDC1* further compress the data using inverse modified discrete cosine transformation.
- The actor *Freq. Inv0* and *Freq. Inv1* compensates for the negation of values by the actors *IDC0* and *IDC1*.
- The final step in MP3 decoding is to synthesise sampled analogue signals represented using *pulse-code modulation*. This is performed by the actors *Synth0* and *Synth1*.

2.7.3 MP3 Playback Application

The SDF graph of an MP3 playback applications is shown in Figure 2.16, adapted from [Wig09, WBS07]. It consists of the following three actors.

- an MP3 decoder (*MP3*),

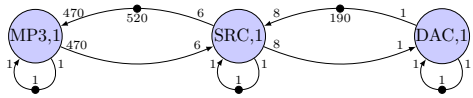


Figure 2.16: SDF graph of an MP3 playback application, adapted from [Wig09, WBS07]

- a sample rate converter (SRC), and
- a digital-to-analogue (DAC) converter.

The MP3 decoder processes the input MP3 file, while digital-to-analogue converter converts the digital output stream from the MP3 decoder to analogue, so that it can be played on an analogue device such as a speaker. However, the sampling rate of both MP3 decoder and digital-to-audio decoder must match, which is performed by the sample rate converter.

2.7.4 Audio Echo Canceller

Audio echo cancellation is an important step in improving voice quality by removing echo or noise present in an audio signal. Figure 2.17 shows the SDF graph of an audio echo canceller used in a mobile phone, adapted from [HGWB12]. It contains the following four actors.

- a sample rate converter (SRC) for matching the sampling rate of the user's voice to the rest of the components in the audio echo canceller,
- an analogue-to-digital (ADC) converter for converting the analogue signal of the echo or noise to a digital format,
- an actual echo canceller (AEC) for separating the user's voice from the echo or noise, and
- output (OUT) which is the echo free voice of the user.

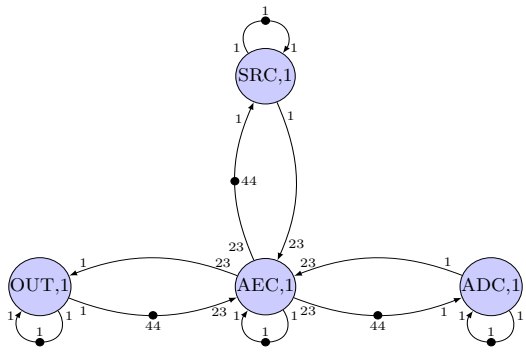


Figure 2.17: SDF graph of an audio echo canceller, adapted from [HGWB12]

2.7.5 Bipartite Graph

Bipartite graph is a well-known and extensively used case study in the literature of SDF analysis [BML99, GBS05]. A bipartite graph is a graph whose vertices can be divided into two disjoint sets, such that every edge connects a vertex from one set to another. That is, there should not be any edge that connects vertices of the same set. The SDF graph containing four vertices (actors) is shown in Figure 2.18, where the actors a and b are in the same set, and the actors c and d are in the same set.

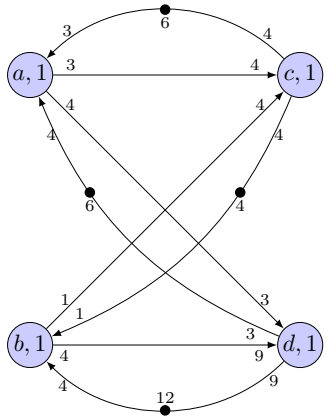


Figure 2.18: Bipartite graph [BML99, GBS05]

2.8 Conclusions

This chapter reports on various notions of dataflow. First we have introduced the need for dataflow models. An important conclusion is that dataflow models better capture parallelism in streaming applications and impose minimal data-dependency constraints, compared to imperative programming languages. Furthermore, dataflow models offer several analytical properties which allow generation of schedules for multiprocessor hardware architectures at design-time.

Moreover, we have formally introduced SDF graphs, and studied different notions associated to SDF graphs, such as auto-concurrency, execution, deadlock, consistency, and throughput. This chapter also has presented the state-of-the-art SDF^3 tool for generating, analysing, and visualising SDF graphs. Furthermore, we also have compared some of the most well-known dataflow models. The comparison shows that the SDF model is best suited for generating schedules and analysing streaming applications at design-time. The chapter ends with presenting several real-life case studies modelled as SDF graphs, which are considered in later chapters.

CHAPTER 3

Model Checking of Timed and Hybrid Automata

STREAMING applications have become an inevitable, integral, and tangible part of a human life, e.g., navigation and video games. With the advancement of hardware technology, streaming applications with more and more complexity are continuously being developed and implemented on embedded multimedia systems, e.g., the support for 3D and VR-based games in mobile phones. Most of the embedded multimedia systems are battery-powered and have limited computational power. Hence, efficient scheduling of streaming applications is very important and significant to obtain robust performance and longer system lifetime. A wide variety of techniques can be applied to achieve this goal, such as classical simulation [Sha75], mathematical optimisation [OPT, Sny05], and model checking [CE81, QS82]. This thesis considers *model checking* for its advantages of guaranteeing optimality and generating schedules.

Model checking is a method for *formal verification* of concurrent systems. It started with two seminal papers, written independently by Clarke and Emerson [CE81] and by Queille and Sifakis [QS82]. In fact, Clarke, Emerson, and Sifakis received 2007 ACM Turing Award for their roles in developing model checking into a highly effective verification technology, widely used in the hardware and software industries [TUR07]. In this thesis, we utilise model checking to derive optimal schedules for streaming applications modelled as SDF graphs mapped on hardware architectures. For this purpose, both SDF graphs and hardware architectures are translated to the model-checking domain.

Chapter 2 introduced SDF graphs and different concepts related to them. In this chapter, we explain different model-checking formalisms which are used later in this thesis. In particular, Timed automata, Priced timed automata, and Hybrid automata are presented, with the help of examples. Moreover, this thesis considers the UPPAAL toolset [LPY97] as a model checking tool for its advantages of generating time- and cost-optimal schedules. This chapter also discusses how timed automata, priced timed automata, and hybrid automata are modelled using UPPAAL.

Chapter Outline. Section 3.1 provides an informal overview of model checking, followed by a formal coverage of timed automata in Section 3.2. Then, priced timed automata is presented in Section 3.3 and Section 3.4 discusses hybrid automata. Finally, Section 3.5 concludes by summarising the contributions of this chapter.

3.1 Model Checking

Model checking operates under the assumption that the system under verification has a finite number of *states*, which can be modified through *state transitions*.

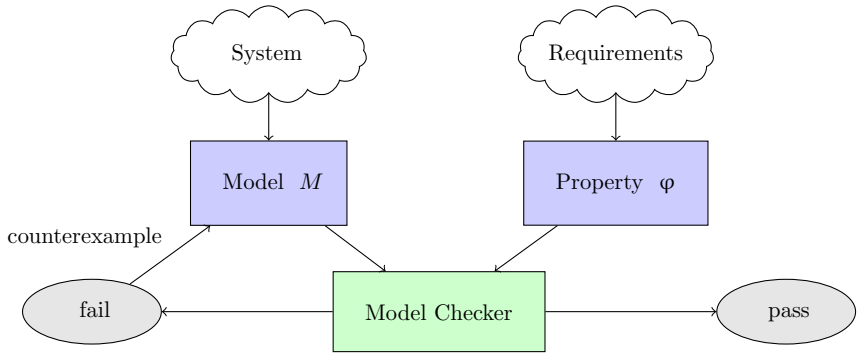


Figure 3.1: An overview of model checking.

Figure 3.1 shows an overview of model checking, taken from [Tim13]. The model checker tool takes as an input a formal description of the system (M), and a formal description of the requirements (φ) which we call a *property*. The model checker then performs exhaustive state-space exploration, and tries to prove the system correctness, or more precisely that the system is a model of its requirements: $M \models \varphi$. However, if the opposite is true, then there exists some state in the system or some execution through the system, that violates the required property. In this case, a nice feature of the model checker is that it delivers a counterexample in the form of an execution trace. The counterexample can be used to improve the system specification.

3.1.1 Temporal Logics for Model Checking

For model checking, model M of the system under consideration needs to be accompanied with a specification of the property of interest φ that is to be verified. *Temporal Logics* [Pnu77] have been deemed a good method for specifying properties of concurrent systems [Lam83]. Temporal logics are generally categorised based on whether the properties they specify are either in the *linear-time* or the *branching-time* domain.

Linear-time domain. Linear temporal logic (LTL) [Pnu77] is termed linear, because time is qualitatively viewed as linear: at each moment of time there is only one possible successor state and thus each time moment has a unique possible future [BK08]. In particular, LTL has operators for saying that a condition over a set of *atomic propositions* will hold *eventually* or that it *always* holds. An extension of LTL is metric temporal logic (MTL) [Koy90] which has real-time metrics and constraints used for the verification of real-time systems.

Branching-time domain. Not all properties are expressible in linear time. For example, in circuit design, the reset property states that "it is always possible to reach a certain *reset* state (even if it is never reached actually)" cannot translated to LTL. LTL can only state that the reset state is *actually* reached, and not that it can be *possibly* reached.

Branching-time logic allows specification of such proprieties by means of existential and universal quantification operators. The most well-known branching-time logic is computation tree logic (CTL) [CE81].

3.1.2 Quantitative Model Checking

Traditionally, model checking only focused on the *qualitative* aspects of the behaviour of a system. For instance, verifying that a certain undesirable event can never occur, e.g., buffer overflowing, or that a certain desirable event is guaranteed to eventually occur, e.g., message arrival.

Over the past two decades, the focus has shifted towards the *quantitative* and *probabilistic* aspects of the behaviour of a system. For example, the *least* amount of time required for a message to arrive, the probability of successful message transmission etc. Using quantitative model checking, real-time systems can be modelled by timed automata (TA) [AD90, AD94] and hybrid automata (HA) [DDL+12]. One novel extension of timed automata is the annotation of models with costs resulting in priced timed automata (PTA) [RTP04, BFH+01b].

Software tools such as the UPPAAL toolset [LPY97], PRISM [KNP11], MRMC [KKZ05, KZH+11], and CADP [GLMS11] are dedicated quantitative model checkers that have been applied to a number of case studies. This thesis utilises the UPPAAL toolset as a model checker because of its distinctive feature of generating time- and cost-optimal traces for *reachability* properties.

Reachability properties. Reachability properties ask whether a given state formula φ can be satisfied by any reachable state. The result of checking such a property is either a trace leading to a state satisfying φ or a message that no reachable state satisfies φ. Reachability properties are often used while designing a model to perform sanity checks. For instance, if we have a model of a communication protocol involving a sender and a receiver, reachability properties can be used to check whether it is possible for the sender to send a message at all or whether a message can possibly be received. We express that some state satisfying φ is reachable using the logical formula $E\Diamond\varphi$. Here, $E\Diamond$ represents that the property φ *eventually* holds in some state of the system.

We specify the scheduling problem of an SDF graph as a reachability property in this thesis, and use the UPPAAL toolset to find out the optimal trace which is translated to a schedule. By UPPAAL toolset, we refer to classical UPPAAL [BDL04], as well as its extensions termed UPPAAL CORA [BLR05] and UPPAAL SMC [DDL+12]. In the UPPAAL toolset, the reachability property is written using the syntax **E** <> φ.

Nondeterminism. Quantitative aspects are often analysed in the presence of *nondeterminism*: unquantified freedom for a system to choose from a set of pos-

sible alternative behaviours [Tim13]. In other words, a system is nondeterministic if at some point the precise behaviour of the system is unknown to us (although the possible alternatives *are* specified). While probabilistic approaches assign the likelihood to each alternative to happen, nondeterminism always leaves the choice open. For example, a video game nondeterministically choosing between respawning the character or aborting the game, after having been killed may always abort the game, respawn if the game is played online and abort otherwise, or do something completely different. Nondeterminism may occur due to an unspecified ordering of events between two or more (partly) independent parallel components, an interaction with an unpredictable environment, or just underspecification.

Scheduler synthesis using nondeterminism. An SDF graph normally exhibits interleaving because any actor that has enough input tokens available can fire. In this thesis, the scheduling choices, i.e., assignment, ordering, and the exact firing time of actors on processors are left nondeterministic for the UPPAAL toolset to resolve. In this way, the UPPAAL toolset searches the whole state-space, and explores each scheduling choice to check whether it is (time- or energy-) optimal. In the end, the UPPAAL toolset generates a trace in which only the optimal scheduling choices are considered, resulting in a (time- or energy-) optimal trace.

Now model checking is introduced, we discuss the following model-checking formalisms. We also explain how these formalisms are modelled using the UPPAAL toolset.

- *Timed automata.* Timed automata (TA) were introduced by Alur and Dill [AD90, AD94] as a natural and versatile model for real-time systems. They extend labelled transition systems with real-valued clocks. TA has the following two major distinctive features.
  - They are able to express realistic constraints using real-valued clocks; and
  - Model checking of TA is decidable [AD90, AD94].

  In this thesis, clocks are used to model the execution time of SDF actors in Chapter 4. Then, UPPAAL is utilised for throughput-optimal scheduling of SDF graphs.

- *Priced timed automata.* Priced timed automata (PTA) extend TA with costs [RTP04, BLR05], which we use to model power consumption of processors in Chapter 5. Afterwards, UPPAAL CORA is utilised for energy-optimal scheduling of SDF graphs.

- *Hybrid automata.* Hybrid automata (HA) extend TA with continuous variables [HR98], which are used to model hybrid behaviour of batteries in Chapter 6. Furthermore, UPPAAL SMC is used to analyse the performance of SDF graphs mapped on battery-powered processors.

  The reachability problems for hybrid automata are known to be undecidable in general [Ras05]. UPPAAL SMC addresses this challenge by offering

statistical model checking approach generalised to handle undecidable problems [BDL+12] .

3.2 Timed Automata

This section introduces the basic definitions of syntax and semantics of timed automata (TA) [AD90, AD94]. In the following, we use *B(C)* to denote a set of clock constraints for a finite set of clocks *C*. That is, *B(C)* contains all conjunctions over simple conditions of the form $x \bowtie c$ or $x - y \bowtie c$, where $x, y \in C$, $c \in \mathbb{N}$ and $\bowtie \in \{<, \leq, =, \geq, >\}$.

3.2.1 Definition

Definition 3.1. *A* timed automaton $\mathcal{A}$ *is a tuple* (L, l^0, Act, C, E, Inv)*, where*

- L *is a set of locations;*
- $l^0 \in L$ *is the initial location;*
- Act *is a finite set of actions, co-actions and internal* λ*-actions;*
- C *is a finite set of clocks;*
- $E \subseteq L \times Act \times B(C) \times 2^C \times L$ *is a set of edges, and*
- $Inv : L \to B(C)$ *assigns an invariant to each location.*

Example 3.2. Figure 3.2 shows an example of a timed automaton of an automatic lamp. Clock y measures the progress of time. The timed automaton of the lamp has three locations, i.e., off, dim and full, where the initial location is off. In the beginning, the lamp moves from off to dim location representing that the lamp is turned on and is emitting dim light. The lamp can stay in dim location for at most 10 time units because of the invariant $y \leq 10$. From here, the lamp can take two actions: either switch off or give full light. If full light is to be emitted, the lamp has to take the edge in less than 5 time units represented by the guard condition $y < 5$. Otherwise, the lamp moves to off location by taking the edge with the guard condition $y \geq 5$. From full location, the lamp switches off automatically between 10 and 15 time units.

3.2.2 Semantics

A clock valuation is a function $\eta : C \to \mathbb{R}_{\geq 0}$ from a set of clock to the non-negative real numbers. Let $\mathbb{R}^C$ be a set of all clock valuations. Edges are labelled with tuples (g, α, D) where g is a clock constraint on the clocks of the timed automaton, α is an action, and $D \subseteq C$ is a set of clocks. We can interpret an edge $l \xrightarrow{g:\alpha,D} l'$ as a timed automaton moving from location l to l' if guard g is satisfied. As a result, an action α is performed and any clock in D is reset to zero. Let $\eta_0(x) = 0$ for all $x \in C$. We will say that the clock values denoted by η satisfy the guard g written as $\eta \models g$. Similarly, the clock values denoted by η satisfies $Inv(l)$, written as $\eta \models Inv(l)$. The semantics of TA are defined below.

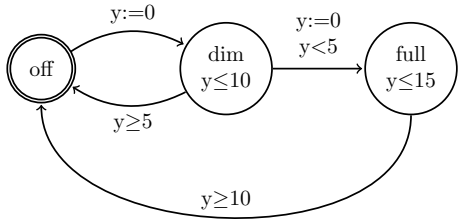


Figure 3.2: Timed automaton of an automatic lamp. The lamp is either off or emits dim or full light.

Definition 3.3. *Let (L, l^0, Act, C, E, Inv) be a timed automaton. The semantics of TA is defined as a labelled transition system $\langle S, s_0, \rightarrow\rangle$ where $S \subseteq L \times \mathbb{R}^C$ is the set of states, $s_0 = (l_0, \eta_0)$, and $\rightarrow \subseteq S \times (\mathbb{R}_{\geq 0} \cup Act) \times S$ is the transition relation such that,*

- $(l, \eta) \xrightarrow{d} (l, \eta + d)$ *if* $\forall d' : 0 \leq d' \leq d \Rightarrow \eta + d' \models Inv(l)$,
  where for $d \in \mathbb{R}_{\geq 0}$, $\eta + d$ maps each clock x in C to the value of $\eta(x) + d$, and
- $(l, \eta) \xrightarrow{a} (l', \eta')$ *if there exists $e = (l, a, g, r, l') \in E$ s.t. $\eta \models g$, $\eta' = [r \mapsto 0]\eta$, and $\eta' \models Inv(l')$,*
  where $[r \mapsto 0]\eta$ denotes the clock valuation which maps each clock in r to 0 and satisfies with η over $C \backslash r$.

Time-critical systems are often modelled as a parallel composition of TA, which is denoted by a parallel composition operator $||$ parametrised with handshaking actions H. Actions in H need to be carried out by both involved timed automata jointly.

Definition 3.4. *Let $\mathcal{A}_i = (L_i, l_i^0, Act_i, C_i, E_i, Inv_i)$, $i = 1, 2$ with $H \subseteq Act_1 \cap Act_2$ and $C_1 \cap C_2 = \varnothing$. The timed automaton $\mathcal{A}_1 || \mathcal{A}_2$ is defined as,*

$$(L_1 \times L_2, l_1^0 \times l_2^0, Act_1 \cup Act_2, C_1 \cup C_2, E, Inv_1 \wedge Inv_2)$$

The edge set E is the smallest set that contains the following transitions:

- *for $\alpha \in H$:*

$$\frac{l_1 \xrightarrow{g_1 : \alpha, D_1}_1 l_1' \wedge l_2 \xrightarrow{g_2 : \alpha, D_2}_2 l_2'}{\langle l_1, l_2\rangle \xrightarrow{g_1 \wedge g_2 : \alpha, D_1 \cup D_2} \langle l_1', l_2'\rangle}$$

- *for $\alpha \notin H$:*

$$\frac{l_1 \xrightarrow{g:\alpha, D}_1 l_1'}{\langle l_1, l_2\rangle \xrightarrow{g:\alpha, D} \langle l_1', l_2\rangle} \quad \textit{and} \quad \frac{l_2 \xrightarrow{g:\alpha, D}_2 l_2'}{\langle l_1, l_2\rangle \xrightarrow{g:\alpha, D} \langle l_1, l_2'\rangle}$$

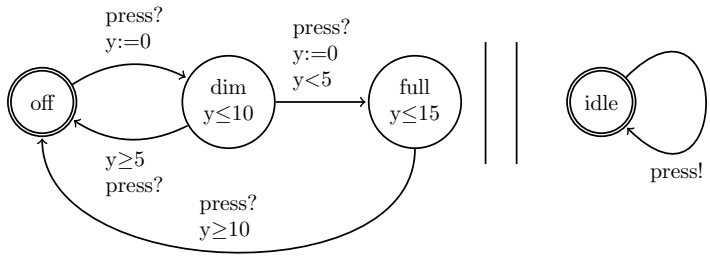


Figure 3.3: Timed automaton of a lamp and user, synchronised on a shared action press. The lamp is either off or emits dim or full light, controlled by the pressing of the switch by the user.

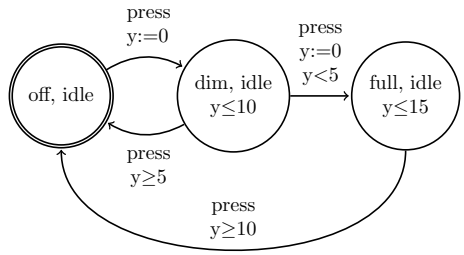


Figure 3.4: Timed automaton $Lamp||_{H=\{press\}} User$

Example 3.5. Figure 3.3 shows the timed automaton in Figure 3.2 parallel composed with user. In this way, the lamp is not automatic anymore, and it is controlled by the user. The timed automaton of the user has one location only, i.e., idle. Both automata synchronise on the action press representing the pressing of the switch to turn the lamp on or off. The complete model system is given by

$$Lamp||_H User$$

where $H = \{press\}$. The composite timed automaton $Lamp||_H User$ is shown in Figure 3.4. The LTS of this example has the following transitions for all $d, t \in \mathbb{R}_{\geq 0}$, where t is a shorthand for the clock valuation $\eta(y) = t$.

$(off, idle, t) \xrightarrow{d} (off, idle, t + d)$ for all $t \geq 0$ and $d \geq 0$

$(off, idle, t) \xrightarrow{press} (dim, idle, 0)$ for all $t \geq 0$

$(dim, idle, t) \xrightarrow{d} (dim, idle, t + d)$ for all $t \geq 0$ and $d \geq 0$ with $t + d \leq 10$

$(dim, idle, t) \xrightarrow{press} (full, idle, 0)$ for all $0 \leq t < 5$

$(dim, idle, t) \xrightarrow{press} (off, idle, t)$ for all $5 \leq t \leq 10$

$(full, idle, t) \xrightarrow{d} (full, idle, t+d)$ for all $t \geq 0$ and $d \geq 0$ with $t + d \leq 15$

$(full, idle, t) \xrightarrow{press} (off, idle, t)$ for all $10 \leq t \leq 15$. □

3.2.3 Timed Automata in UPPAAL

UPPAAL supports additional syntax for convenient modelling of TA. In particular, UPPAAL models can declare variables that can be used in guards, and be updated on edges. This subsection explains the related features extended to TA by UPPAAL modelling language, taken from [BDL04].

- A *system model* in UPPAAL consists of a network of processes. The description of a model has three parts, i.e., declarations, automata templates and system definition.
- *Declarations* are either local or global and may contain declarations of clocks, arrays, (bounded) integers, channels and types. For example,
  - const int a = 1; represents constant a with value 1 of type integer.
  - bool b[8], c[4]; represent two boolean arrays b and c, with 8 and 4 elements respectively.
  - int[0, 100] a = 5; denotes an integer variable a with the range [0, 100] initialised to 5.
  - int a[2][3] = {{1, 2, 3}, {4, 5, 6}}; denotes a multidimensional integer array a with default range and an initialiser. Arrays are permitted for clocks, channels, integer variables and constants.
  - clock x, y; denote two clocks x and y.
  - chan d; represents a channel d.
  - urgent chan e; represents a urgent channel e.
  - struct {int a; bool b;} s1 = {2, true}; denote an instantiation of a structure where the members a and b are set to 2 and true.
  - Custom types are defined with the C-like *typedef* construct. For example, the following declares a structure type A containing an integer a, and a boolean b:
    typedef struct
    {
    int a;
    bool b;
    } A;
  - meta int swap;
    int a;
    int b;
    assign swap = a; a = b; b = swap;
    express a meta variable used to swap the contents of two integers a and b.

Integers, booleans, and arrays and records over integers and booleans can be marked as *meta variables* by prefixing the type with the keyword meta. Meta variables are stored in the state vector, but are semantically not considered part of the state, i.e., two states that only differ in meta variables are considered to be equal.

- *Templates*, which are blueprints of TA, have local declarations and a set of parameters of any type, e.g., *int*, *chan*. A template is instantiated in the system definition.

- In the *system definition*, the whole system model is defined in terms of one or more concurrent processes (instances of templates).

- Automata synchronise through *channels*. *Binary* channels model binary and, are declared as chan c. An edge labelled as c! denotes a sender and synchronises with another edge labelled as c? representing a receiver.

- *Broadcasting* channels model asymmetric one-to-many synchronisation and are declared as broadcast chan c. In a broadcast channel, one sender c! can synchronise with an arbitrary number of receivers c?. Note that broadcasting channels are non-blocking.

- *Urgent* locations are semantically equivalent to adding an extra clock x, that is reset on all incoming edges, and having an invariant $x \leq 0$ on the location. Hence, time is not allowed to pass when the system is in an urgent location.

- *Committed* locations are even more restrictive on the execution than urgent locations. A committed location cannot delay and the next transition must involve an edge from one of the committed locations.

- *User defined functions* are defined either globally or locally to the templates. Local functions can access the template parameters.

Expressions in UPPAAL can be written over clocks and variables and are placed using the following *labels*. All of these expressions are associated to edges except *invariants* which are associated to *locations*.

- A *select* label contains a comma-separated list of name : type expressions where name is a variable name and type is a defined type. The select label nondeterministically binds name to a value in the range of type.

- *Guards* are side-effect free expressions on edges and evaluates to a boolean. Only clocks, integer variables and constants can be referenced. Guards over clocks are essentially conjunctions of clock bounds. An edge annotated with a guard is enabled in a state if and only if the guard evaluates to true.

- Processes can synchronise over channels. A *synchronisation* label is of a form Expression! or Expression? or can be empty. A synchronisation label must be side-effect free.

- When executed, the *update* expression of the edge is evaluated. An update is a comma-separated list of expressions with side-effects. Expressions in an update label must refer to clocks, integers, variables and constants only. They may also call functions.

- An *invariant* is a side-effect free label which can refer to clocks, integers, variables and constants only. An invariant is a conjunction of conditions of the form x < e or x <= e where x is a clock and e evaluates to an integer.

The UPPAAL interface has three tabs, i.e., the editor, the simulator and the verifier. The key idea is that the user models a system graphically in the editor, simulates it to check its behaviour, and verifies it in the verifier against a set of queries.

UPPAAL also offers the following options for generating a counterexample or witness trace (if there is one) during verification. The trace is loaded into the simulator after verification. Traces can be of the following types.

- *Shortest.* Generate a shortest trace, i.e., a trace with the smallest number of transitions.

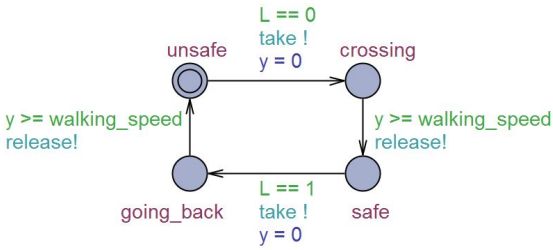
- *Fastest.* Generate a fastest trace, i.e. a trace with the shortest accumulated time delay.

In this thesis, we are interested in generating *fastest* traces, as they represent throughput-optimal schedules.

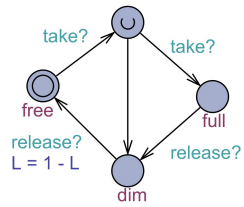
Example 3.6. Consider a bridge puzzle (adapted from the UPPAAL distribution) having four soldiers $s1$, $s2$, $s3$, and $s4$ who need to safely cross a river via the bridge at night. There is only one flash light available. As the bridge is completely dark, it is not possible to cross the bridge without the flash light. There cannot be more than two soldiers on the bridge at a same time. Furthermore, each soldier has a different walking speed, i.e., $s1$, $s2$, $s3$, and $s4$ takes 5 min, 10 min, 20 min, and 25 min respectively. The flash light has two levels of producing light, i.e., dim or full. If one soldier is on the bridge, dim light is sufficient. If there are two soldiers, it is important to have full light. Otherwise, the flash light is off.

Figure 3.5a shows the UPPAAL model of a soldier. This automaton has four locations, i.e., unsafe, crossing, safe, and going_back representing if the soldier has not crossed the bridge, is currently crossing the bridge, has safely crossed the bridge, or is going back to return the flash light respectively. The variable walking_speed is a parameter representing the walking speed of each soldier. The clock variable y records progress of time. As there are four soldiers who need to cross the bridge, this automaton is instantiated four times.

Figure 3.5b shows the UPPAAL model of the flash light. The flash light can either be off, or emit dim or full light depending on the number of soldiers on the bridge. Both automata synchronise on the channels take and release. The integer variable L represents if the flash light is present either at the safe or unsafe side of the bridge. If the value of L is 0, it is present at the unsafe side of the bridge. Otherwise, it is present at the safe side. Initially, L is 0.



(a) UPPAAL model of a soldier who wants to cross a bridge. Each soldier can take the flash light via the channel take, and release it after crossing the bridge via the channel release. The variable walking_speed is a parameter representing the walking speed of each soldier.



(b) UPPAAL model of the flash light capable of producing dim or full light.

Figure 3.5: UPPAAL models of soldier and flash light

Consider a soldier on the unsafe side of the bridge, i.e., location unsafe in Figure 3.5a. If the flash light is also present on the same side, the soldier can take the flash light via the channel take, and move to the location crossing representing that the soldier is now crossing the bridge. The flash light automaton moves to the location marked U representing that this location is urgent. In urgent locations, time is not allowed to progress. If there is another soldier who wants to cross the bridge, the flash light automaton moves to the location full by synchronising on the channel take. This represents the fact that there are two soldiers on the bridge, and thus full light is required. Otherwise, the flash light automaton moves to the location dim representing the fact that there is only soldier on the bridge, and dim light is sufficient.

Consider there are two soldiers on the bridge, and the flash light automaton is in the location full. After a soldier crosses the bridge, then the soldier automaton moves to the location safe from the location crossing, by synchronising on the channel release. As a result, the flash light automaton moves to the location dim representing that currently there is only one soldier on the bridge. The edge from the location crossing to the location safe is annotated with a guard condition $y \geq$ walking_speed representing the walking speed of the soldier.

From the location dim, the flash light automaton can only move to the location free by synchronising on the channel release. As a result, the second soldier also crosses the bridge safely and the respective automaton moves to the location safe. The value of L changes to 1, denoting that the flash light is on the safe side. However, the two remaining soldiers on the unsafe side also need the flash light to cross the bridge. Thus, one of the safe soldiers needs to go back.

The scheduling problem in which we are interested is what is the minimum time for everyone to cross the bridge. In UPPAAL, we use the following query and ask for the fastest trace.

$$\mathbf{E} <> (\texttt{soldier1.safe and soldier2.safe and soldier3.safe and soldier4.safe}) \qquad \square$$

Figure 3.6: Time-optimal schedule of the soldiers for crossing the bridge. The white and red blocks denote if the soldier is moving from the unsafe side to the safe side or vice versa respectively.

Here, `soldier1` represents the name of the automaton modelling the soldier $s1$, and so on.

Figure 3.6 shows a time-optimal schedule of the soldiers for crossing the bridge. First the soldiers $s1$ and $s2$ cross the bridge, and the soldier $s1$ comes back with the flash light. Afterwards the soldiers $s3$ and $s4$ cross the bridge, and the soldier $s2$ returns with the flash light. Finally the soldiers $s1$ and $s2$ cross the bridge. The whole process takes 60 minutes.

3.3 Priced Timed Automata

Price timed automata (PTA) extend TA with costs [RTP04, BLR05]. Costs can either be accumulated in states, proportionally to the residence time, or by taking an edge. Similar to TA, PTA can be analysed for a wide number of properties, including absence of deadlocks, safety, and liveness.

3.3.1 Definition

Definition 3.7. *A* priced timed automaton $\mathcal{PT}$ *over clocks* C *and actions* Act *is a tuple* $(L, l^0, Act, E, \mathcal{P}, Inv)$*, where*

- L *is a set of locations;*
- $l^0 \in L$ *is the initial location;*
- Act *is a finite set of actions, co-actions and internal* λ*-actions;*
- $E \subseteq L \times Act \times B(C) \times 2^C \times L$ *is a set of edges;*
- $\mathcal{P} : (L \cup E) \to \mathbb{N}$ *assigns costs to edges and locations, and*
- $Inv : L \to B(C)$ *assigns an invariant to each location.*

Example 3.8. Figure 3.7 extends the example in Figure 3.3 with costs, making it a PTA model. For simplicity, the invariants are removed. The costs are used to model power consumption of the lamp. The priced timed-automaton of the lamp has three locations, i.e., off, dim and full. Initially, the lamp is in the off location. The PTA of the user contains one location only, i.e., idle. If the user presses a switch once and synchronises with press, then the lamp is on and emits dim light, while consuming power equal to 4 W (indicated by the differential equation $\mathsf{p}' == 4$). The user has to press again to switch off the lamp, or to get

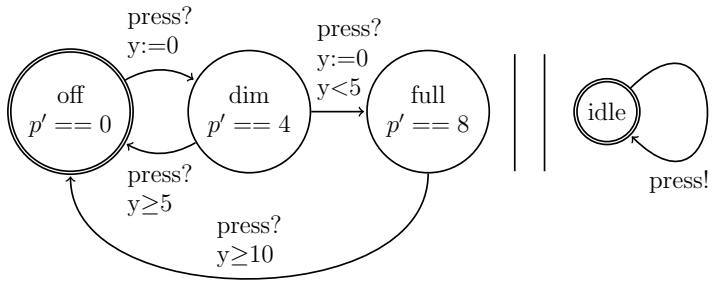


Figure 3.7: Priced timed automaton of a lamp and user, modelled by extending Figure 3.3 with costs. The costs are used to model the power consumption of the lamp. The user can choose the lamp to be off, or to emit dim or full light. The lamp consumes no power while off. Otherwise, a higher amount of light requires a higher power consumption.

full light. If full light is required, the switch must be pressed rapidly (in less than 5 time units indicated by the condition y <5), in which case the lamp consumes more power (8 W). To switch off the lamp, the user has to wait at least 5 time units (indicated by the condition y ≥5). The clock y is used to detect if user is fast (y <5) or slow (y ≥5). The cost variable is represented by p.

3.3.2 Semantics

Clock values in PTA like TA are represented as a function termed clock valuations from C to the non-negative reals $\mathbb{R}_{\geq 0}$. Let $\mathbb{R}^C$ be a set of all clock valuations. The semantics of PTA are defined as a priced transition system [BFH+01a]. A priced transition system is a labelled transition system, where the transition relation is given by a partial function from transitions to the non-negative reals, intuitively being the cost of the transition. We write $l \xrightarrow{g:\alpha,D}{}^{p} l'$ whenever the function is defined on the edge (g, α, D) and the cost is p.

Definition 3.9. *Let $(L, l^0, Act, E, \mathcal{P}, Inv)$ be a priced timed automaton. The semantics of PTA is defined as a labelled transition system $\langle S, s_0, \Sigma, \rightarrow\rangle$ where $S \subseteq L \times \mathbb{R}^C$ is the set of states, $s_0 = (l_0, \eta_0)$, $\Sigma = \mathbb{R}_{\geq 0} \cup Act$ is the set of labels, and $\rightarrow : (S \times \Sigma \times S) \nrightarrow \mathbb{R}_{\geq 0}$ is a partial function from transitions to the non-negative reals representing the transition relation defined below.*

- $(l, \eta) \xrightarrow{d}{}^{p} (l, \eta + d)$ *if* $\forall d' : 0 \leq d' \leq d \Rightarrow \eta + d' \models Inv(l)$, *and* $p = d \cdot \mathcal{P}(l)$, *where for* $d \in \mathbb{R}_{\geq 0}$, $\eta + d$ *maps each clock* x *in* C *to the value of* $\eta(x) + d$, *and*
- $(l, \eta) \xrightarrow{a}{}^{p} (l', \eta')$ *if there exists* $e = (l, a, g, r, l') \in E$ *s.t.* $\eta \models g$, $\eta' = [r \mapsto 0]\eta$, $\eta' \models Inv(l')$, *and* $p = \mathcal{P}(l, a, g, r, l')$,
  where $[r \mapsto 0]\eta$ *denotes the clock valuation which maps each clock in* r *to 0 and satisfies with* η *over* $C \backslash r$.

For PTA that are synchronised over their set of actions H, parallel composition is defined as follows [PSE04].

Definition 3.10. *Consider two priced timed automaton* $\mathcal{PT}_i = (L_i, l_i^0, Act_i, E_i, \mathcal{P}_i, Inv_i)$, $i = 1, 2$ *with* $H \subseteq Act_1 \cap Act_2$ *and* $C_1 \cap C_2 = \varnothing$. *The PTA* $\mathcal{PT}_1 || \mathcal{PT}_2$ *is defined as,*

$$(L_1 \times L_2, l_1^0 \times l_2^0, Act_1 \cup Act_2, E, \mathcal{P}_1 + \mathcal{P}_2, Inv_1 \wedge Inv_2)$$

The edge set E *is the smallest set that contains the following transitions:*

- *for* $\alpha \in H$:

$$\frac{l_1 \xrightarrow{g_1:\alpha, D_1}_1 l_1' \wedge l_2 \xrightarrow{g_2:\alpha, D_2}_2 l_2'}{\langle l_1, l_2 \rangle \xrightarrow{g_1 \wedge g_2:\alpha, D_1 \cup D_2} \langle l_1', l_2' \rangle}$$

- *for* $\alpha \notin H$:

$$\frac{l_1 \xrightarrow{g:\alpha, D}_1 l_1'}{\langle l_1, l_2 \rangle \xrightarrow{g:\alpha, D} \langle l_1', l_2 \rangle} \quad \textit{and} \quad \frac{l_2 \xrightarrow{g:\alpha, D}_2 l_2'}{\langle l_1, l_2 \rangle \xrightarrow{g:\alpha, D} \langle l_1, l_2' \rangle}$$

The cost of an execution trace is simply the accumulated cost of all transitions in the trace, as defined in the following [BFH+01a].

Definition 3.11. *Let* $\tau = (l_0, \eta_0) \xrightarrow{a_1}^{p_1} (l_1, \eta_1) \ldots \xrightarrow{a_n}^{p_n} (l_n, \eta_n)$ *be a finite execution trace. The cost of* τ, *cost*(τ), *is the sum* $\sum_{i=1}^{n} p_i$. *For a given state* (l, η), *the minimum cost mincost*(l, η) *of reaching the state, is the infimum of the costs of finite traces ending in* (l, η). *For a given location* l, *the cost-optimal reachability problem is to find the largest cost* k *such that* $k \leq mincost(l, \eta)$ *for all clock valuations* η.

Example 3.12. The complete model of the system in Figure 3.7 is given by

$$Lamp ||_H User$$

where $H = \{press\}$. The composite timed automaton $Lamp ||_H User$ is shown in Figure 3.8. The transitions in the LTS of this example remain the same as

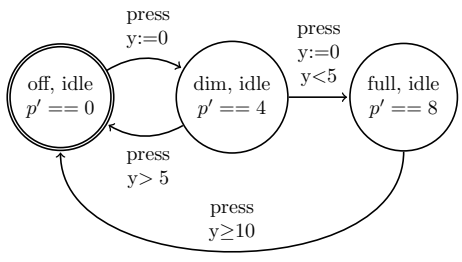
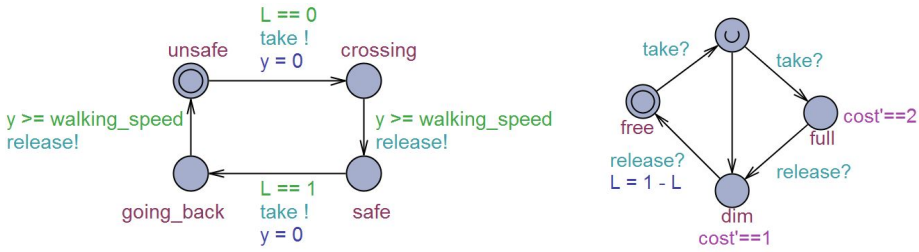


Figure 3.8: Timed automaton $Lamp ||_{H=\{press\}} User$



(a) UPPAAL CORA model of a soldier

(b) UPPAAL CORA model of the flash light which consumes power when turned on

Figure 3.9: UPPAAL models of soldier and flash light

in Figure 3.4. A sample trace τ_0 is given below. The accumulated cost of the following trace is $cost(\tau_0) = 88$.

$$\tau_0 = (off, idle, 0) \xrightarrow{press}{}^{0} (dim, idle, 0) \xrightarrow{2}{}^{8} (dim, idle, 2)$$
$$\xrightarrow{press}{}^{0} (full, idle, 0) \xrightarrow{10}{}^{80} (full, idle, 10) \xrightarrow{press}{}^{0} (off, idle, 10) \qquad \square$$

3.3.3 Priced Timed Automata in UPPAAL CORA

UPPAAL CORA is a branch of UPPAAL which has a convenient support for modelling PTA and finding cost-optimal schedules. Optimality is defined in terms of a variable named cost. The optimal trace can be found by using the *Best trace* option. With this option, UPPAAL CORA keeps searching until a trace to a goal state with the smallest value for the cost variable has been found. The rate of growth of cost is specified as cost′. UPPAAL CORA inherits all other features from UPPAAL. It is worth mentioning that in general, there can be more than one cost variables in PTA. However, in UPPAAL CORA, there can be only one cost variable.

Example 3.13. Consider the example in Figure 3.5. We extend this example in such a way that the flash light runs on a battery. The flash light consumes power equal to 1 and 2 W when producing dim and full light respectively. Figure 3.9 shows the UPPAAL CORA model of this example. The power consumption of the flash light is modelled using the variable cost′ in full and dim locations in Figure 3.9b. The soldier automaton remains the same.

Now the scheduling problem is: what is the minimum energy consumption of the flash light for everyone to cross the bridge? We use the same query as used in UPPAAL but ask for the best trace.

Figure 3.10 shows an energy-optimal schedule of this example. We can see that the soldier $s1$ has to return every time with the flash light so that other soldiers can cross the bridge. This schedule takes 65 minutes and the total energy consumption is 80 W·minutes.

Figure 3.10: Energy-optimal schedule of the soldiers for crossing the bridge. The white and red blocks denote if the soldier is moving from the unsafe side to the safe side or vice versa respectively.

The schedule in Figure 3.10 is energy-optimal but not time-optimal. We have seen in Figure 3.6 that the minimum time required by everyone to cross the bridge is 60 minutes. We can ask for an energy- and time-optimal schedule by adding a global clock variable time which is never reset. The following query is then used. The schedule remains the same as Figure 3.6.

$$\mathbf{E} <> (\texttt{soldier1.safe and soldier2.safe and soldier3.safe and soldier4.safe and time} <= 60) \quad \square$$

3.4 Hybrid Automata

3.4.1 Definition

Hybrid automata (HA) extend TA with continuous variables [HR98]. Let X be a finite set of continuous variables. A variable valuation over X is a mapping $v : X \to \mathbb{R}$, where $\mathbb{R}$ is the set of reals. We write $\mathbb{R}^X$ for the set of valuations over X. Valuations over X evolve over time according to delay functions $F : \mathbb{R}_{\geq 0} \times \mathbb{R}^X \to \mathbb{R}^X$, where for each delay d and valuation v, $F(d, v)$ provides the new valuation after a delay of d. It is worth mentioning that in UPPAAL SMC, the delay function F allows the continuous variables X to evolve according to differential equations.

Definition 3.14. *A* hybrid automaton $\mathcal{H}$ *is a tuple* $(L, l^0, Act, X, E, F, Inv)$, *where*

- L *is a finite set of locations;*
- $l^0 \in L$ *is the initial location;*
- Act *is a finite set of actions, co-actions and internal* λ*-actions;*
- X *is a finite set of continuous variables;*
- E *is a finite set of edges in the form* (l, g, a, φ, l'), *where* l *and* l' *are locations,* g *is a predicate on* $\mathbb{R}^X$, $a \in Act$ *is an action label, and* φ *is a binary relation on* $\mathbb{R}^X$*;*
- *for each location* $l \in L$, $F(l)$ *is a delay function defined as* $F : \mathbb{R}_{\geq 0} \times \mathbb{R}^X \to \mathbb{R}^X$*; and*

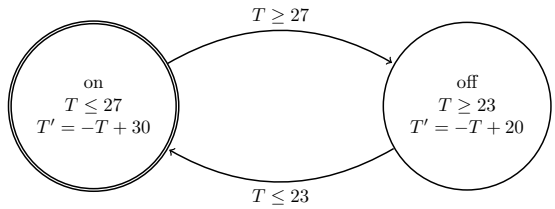


Figure 3.11: Hybrid automaton of a thermostat maintaining the temperature T of a room at 25 Celsius

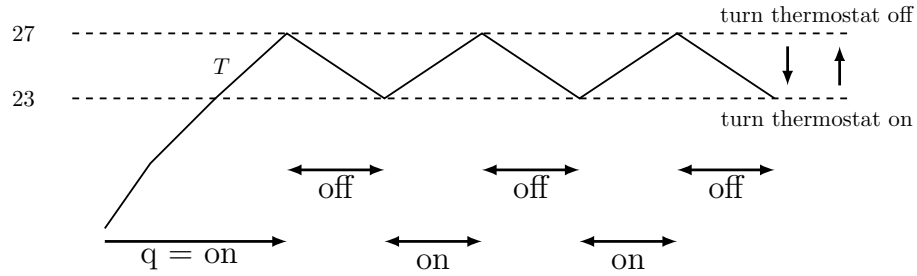


Figure 3.12: Evolution of the continuous and the discrete states of the hybrid automaton in Figure 3.11

- *Inv assigns an invariant predicate Inv(l) to any location l.*

Example 3.15. Let us consider an example of a thermostat maintaining the temperature T of a room at 25 Celsius. If the thermostat is on, the temperature dynamics is given by $T' = -T + 30$, and if it is off, the temperature dynamics is given by $T' = -T + 20$. The hybrid automaton describing the heating of the room is shown in Figure 3.11. The two states $q(t) \in \{\mathsf{on}, \mathsf{off}\}$ represent the two discrete modes of the system: the thermostat is either *on* or *off*. As long as the thermostat is in on mode, the temperature T will follow the dynamics specified in the left state, i.e., T will tend to 30. When the temperature is 27, the thermostat jumps from on to off mode. This is indicated by the invariant in on mode and the guard condition on the transition from on to off mode. In off mode, the temperature follows the dynamics given by the differential equation specified in the right location, i.e., T will tend to 20. When the temperature is 23, the thermostat jumps from off to on mode. This is indicated by the invariant in off mode and the guard condition on the transition from off to on mode.

The state evolution of this example is shown in Figure 3.12. Initially the temperature is $T = 0$, and the thermostat is in the mode $q = \mathsf{on}$. Furthermore, the thermostat is switched on and off via the discrete jumps at $T = 23$ and 27 respectively. □

3.4.2 Semantics

We write $l \xrightarrow{g:\alpha,\varphi} l'$ whenever the automaton can move from location l to l' if guard g is satisfied such that $\upsilon \models g$ and $\varphi(\upsilon, \upsilon')$ for some valuation υ'. As a result, an action α is performed. The semantics of HA are defined below [DDL+13].

Definition 3.16. *Let $(L, l^0, Act, X, E, F, Inv)$ be a hybrid automaton. The semantics of HA is defined as a labelled transition system $\langle S, s_0, \rightarrow \rangle$ where $S \subseteq L \times \mathbb{R}^X$ is the set of states, $s_0 = (l_0, \upsilon_0)$, and $\rightarrow \subseteq S \times (\mathbb{R}_{\geq 0} \cup Act) \times S$ is the transition relation such that,*

- $(l, \upsilon) \xrightarrow{d} (l, \upsilon')$ *with* $d \in \mathbb{R}_0$ *and* $\upsilon' = F(l)(d, \upsilon)$, *and*
- $(l, \upsilon) \xrightarrow{a} (l', \upsilon')$ *if there exists* $e = (l, g, a, \varphi, l') \in E$ *s.t.* $\upsilon \models g$ *and* $\varphi(\upsilon, \upsilon')$.

For HA that are synchronised over their set of actions H, parallel composition is defined as follows [DDL+12].

Definition 3.17. *Let $\mathcal{H}_i = (L_i, l_i^0, Act_i, X_i, E_i, F_i, Inv_i)$, $i = 1, 2$ with $H \subseteq Act_1 \cap Act_2$ and $X_1 \cap X_2 = \varnothing$. The hybrid automata $\mathcal{A}_1 || \mathcal{A}_2$ is defined as,*

$$(L_1 \times L_2, l_1^0 \times l_2^0, Act_1 \cup Act_2, X_1 \cup X_2, E, F, Inv_1 \wedge Inv_2)$$

where $F(l)(d, \upsilon)(x) = F_i(l_i)(d, \upsilon \downarrow_{X_i})(x)$ when $x \in X_i$. Here, $\upsilon \downarrow_{X_i}$ is the projection of υ to X_i if $\upsilon \in \mathbb{R}^X$ with $X = X_1 \cup X_2$. The edge set E is the smallest set that contains the following transitions:

- *for $\alpha \in H$:*

$$\frac{l_1 \xrightarrow{g_1:\alpha,\varphi_1}_1 l_1' \wedge l_2 \xrightarrow{g_2:\alpha,\varphi_2}_2 l_2'}{\langle l_1, l_2 \rangle \xrightarrow{g_1 \wedge g_2:\alpha,\varphi_1 \cup \varphi_2} \langle l_1', l_2' \rangle}$$

- *for $\alpha \notin H$:*

$$\frac{l_1 \xrightarrow{g:\alpha,\varphi}_1 l_1'}{\langle l_1, l_2 \rangle \xrightarrow{g:\alpha,\varphi} \langle l_1', l_2 \rangle} \quad \textit{and} \quad \frac{l_2 \xrightarrow{g:\alpha,\varphi}_2 l_2'}{\langle l_1, l_2 \rangle \xrightarrow{g:\alpha,\varphi} \langle l_1, l_2' \rangle}$$

Example 3.18. The LTS of the thermostat in Figure 3.11 has the following transitions for $d \in \mathbb{R}_0$.

$(on, T) \xrightarrow{d} (on, T)$ for all $d \geq 0$ and $T \geq 0$ with $T \leq 27$

$(on, T) \rightarrow (off, T)$ for all $T \geq 0$ with $T = 27$

$(off, T) \xrightarrow{d} (off, T)$ for all $d \geq 0$ and $T \geq 0$ with $T \geq 23$

$(off, T) \rightarrow (on, T)$ for all $T \geq 0$ with $T = 23$ □

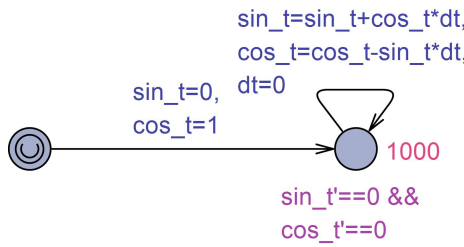


Figure 3.13: UPPAAL SMC model of sine and cosine functions (taken from [BDL+12])

3.4.3 Hybrid Automata in UPPAAL SMC

HA can be analysed by UPPAAL SMC, which implements the delay function F using invariants on clock variables with differential equations of the form $\mathtt{x}' == \mathtt{e}$, where $\mathtt{x}$ is a clock variable and $\mathtt{e}$ evaluates to an integer. Moreover, UPPAAL SMC employs statistical model checking in which several simulation runs of the system with respect to some property are monitored, and then results from statistics are used to get an overall estimate of the value of a property. The values of expressions (evaluating to integers or clocks) can be visualised along the simulation runs in the form of a line or bar plot, by using the following query.

$$\textbf{simulate N } [<= \mathtt{bound}]\{\mathtt{E1}, \ldots, \mathtt{Ek}\}$$

where N is a natural number representing the number of simulations to be performed, *bound* is a time bound on the simulations, and $E1, \ldots, Ek$ are k state-based expressions that are to be monitored and visualised. UPPAAL SMC also supports the evaluation of expected values of min or max of an expression that evaluates to integers or clocks. The syntax of the queries is as follows.

$$\mathbf{E}[\mathtt{bound}; \mathtt{N}]\ (\textbf{min} : \mathtt{expr})$$

or

$$\mathbf{E}[\mathtt{bound}; \mathtt{N}]\ (\textbf{max} : \mathtt{expr})$$

where *bound* is a time bound on the runs, N is the explicit number of runs, and *expr* is the expression to be evaluated.

Example 3.19. Figure 3.13 shows a UPPAAL SMC model of sine and cosine functions [BDL+12]. The clocks sin_t and cos_t are used to compute $\sin(t)$ and $\cos(t)$ using the following facts.

- $\sin(t + dt) \approx \sin(t) + \sin'(t)dt$ for small steps of $dt \to 0$, whereas $\sin'(t) = \cos(t)$ and $\sin(0) = 0$, and
- $\cos(t + dt) \approx \cos(t) - \cos'(t)dt$ for small steps of $dt \to 0$, whereas $\cos'(t) = \sin(t)$ and $\cos(0) = 1$.

The high exponential rate (1000) tells the UPPAAL SMC engine to take small (random) time steps and record the duration in clock dt. The value of variables

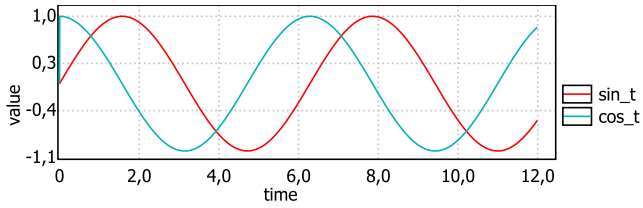


Figure 3.14: Evolution of variables sin_t and cos_t in terms of time

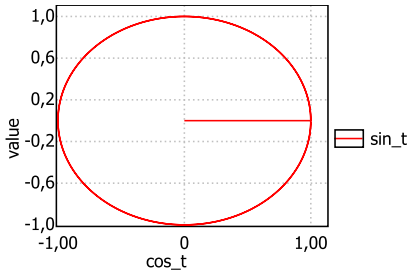


Figure 3.15: sin_t values versus cos_t values

sin_t and cos_t in terms of time is plotted in Figure 3.14, using the following query.

$$\textbf{simulate 1 } [<= 12]\{\texttt{sin\_t}, \texttt{cos\_t}\}$$

Similarly, Figure 3.15 shows sin_t values with corresponding cos_t which forms a circle. The following query has been used for this purpose.

$$\textbf{simulate 1 } [\texttt{cos\_t} <= 1]\{\texttt{sin\_t}\}$$

The expected values of min or max of sin_t and cos_t can also be evaluated using UPPAAL SMC. For example, the queries

$$\textbf{E}[<= 12; 10]\ (\textbf{min} : \texttt{sin\_t})$$

and

$$\textbf{E}[<= 12; 10]\ (\textbf{max} : \texttt{sin\_t})$$

evaluates to -1 and 1 respectively. □

3.5 Conclusions

This chapter has introduced different concepts related to model checking. This chapter has further defined and discussed different model-checking formalisms, i.e., timed automata, priced timed automata, and hybrid automata with the help of examples. The timed automaton model combines real-time clocks with nondeterministic choices. We will use clocks in Chapter 4 to model execution times of actors, and nondeterministic choices for throughput-optimal scheduling

of SDF graphs. PTA extend TA with costs, which we will use in Chapter 5 to model power consumption of processors. This enables us to perform energy-optimal scheduling of SDF graphs. HA extend TA with continuous variables which we use to model hybrid behaviour of batteries in Chapter 6. We also explain how these formalisms are modelled using the UPPAAL toolset in this chapter.

Part II

Scheduling and Analysis

CHAPTER 4

Resource-Constrained Scheduling

Abstract

SYNCHRONOUS dataflow (SDF) graphs are a widely used formalism for modelling, analysing and realising streaming applications, both on a single processor and in a multiprocessing context. Efficient scheduling methods are essential to obtain maximal throughput under the constraint of the available number of resources. This chapter presents an approach to schedule SDF graphs using the proven formalism of *timed automata* (TA). TA maintain a good balance between expressiveness and tractability, and are supported by powerful verification tools, e.g., UPPAAL. In this chapter, we describe a compositional translation of SDF graphs to TA, and analysis and verification in the state-of-the-art tool UPPAAL. This approach does not require the (exponential) transformation of SDF graphs to homogeneous SDF graphs and helps to find schedules with a compromise between the number of processors required and the throughput. This translation also forms the basis to extend this analysis of SDF graphs with new features such as energy consumption and batteries in later chapters.

About this chapter: The current chapter is based on the paper "Resource-Constrained Optimal Scheduling of Synchronous Dataflow Graphs via Timed Automata", which was published at ACSD 2014 [AdGH+14a]. An extended report on the work was published at University of Twente Eprints [AdGH+14b]. The original paper largely remains the same.

4.1 Introduction

Modern multimedia applications, such as multi-party video conferencing and video-in-video, impose high demands on the system throughput. At the same time, resource requirements, e.g., buffer sizes and number of processors used should be minimised. Therefore, smart scheduling strategies are needed.

Synchronous Dataflow (SDF) graphs are well-known computational models for analysing dataflow and digital signal processing applications. Nowadays, they are increasingly utilised for both modelling and analysing multimedia applications on a multiprocessor systems [LM87b, SB09, BSLM96]. Currently, resource-allocation strategies and scheduling of tasks for SDF graphs are carried out using the max-plus algebraic semantics and graph analysis by transforming SDF graphs to equivalent homogeneous SDF graphs (HSDF) [dKBS12, Kar78]. This approach leads to a larger graph; in the worst case, the derived HSDF graph can be exponentially larger than the original SDF graph [DSB+11]. Another state-of-the-art method [GGS+06] calculates the throughput of SDF graphs by exploring the state-space until a periodic phase is found. However, this method executes each task as soon as it is enabled and it is assumed that sufficient resources are available to accommodate all the enabled executions simultaneously. On the contrary, it may not be the case in the real-life applications, where there is always a constraint on the number of resources.

We propose an alternative, novel approach to analyse schedules of SDF graphs on a limited number of processors using timed automata (TA) [AD94]. TA are a natural choice for modelling time-critical systems and to check whether all timing constraints are met. By definition, TA are automata in which clock variables measure the elapse of time. *Clock guards* on the edges indicate conditions under which an edge can be taken and *invariants* show the conditions under which a system can stay in a certain location. TA are extensively used in the verification and model checking of industrial applications [NM10, TY98, BGK+02, SS01].

The complete overview of our approach is shown in Figure 4.1. We translate SDF graphs and hardware architecture to TA using the model checker UPPAAL. After defining the mapping of actors to processors, we utilise UPPAAL to search the state-space and derive a schedule that fits on the given number of processors and maximises throughput. In this way, we can efficiently determine a trade-off between the number of processors and throughput for a certain application. We also demonstrate that our translation preserves deadlock freedom if the number of processors varies.

The main contributions of this chapter are:

- Deriving a schedule that fits on the given number of processors and maximises throughput;
- Handling heterogeneous processor models, in which only specific processors can run a particular task due to their computational limitations; and
- Determining a trade-off between the number of processors and throughput.

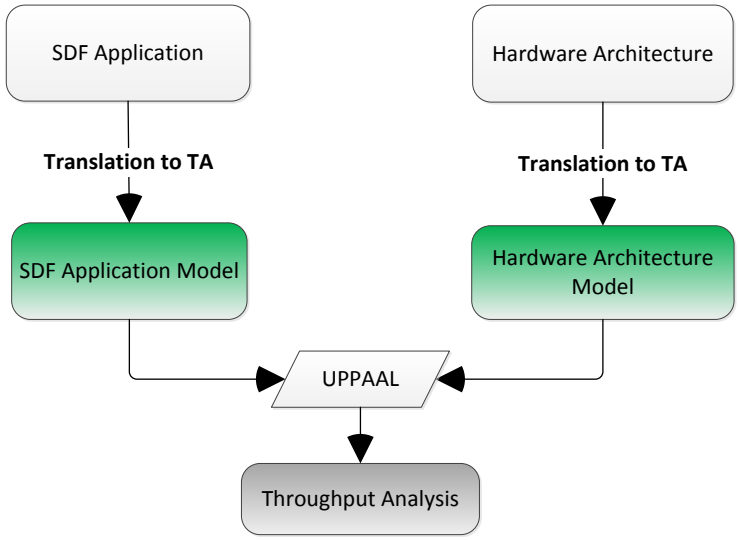


Figure 4.1: Our approach using timed automata. After both SDF application and hardware architecture are modelled as timed automata, UPPAAL is used to derive throughput-optimal schedules.

Chapter Outline. Firstly, Section 4.2 reviews related work. Section 4.3 explains how we extend SDF graphs with resource constraints, and Section 4.4 describes the throughput analysis of resource-constrained SDF graphs. The translation of SDF graphs and platform models to TA using UPPAAL is explained in Section 4.5. Section 4.6 discusses the throughput analysis of SDF graphs using UPPAAL, and Section 4.7 experimentally validates our approach via case studies. The tool-chain developed to support this work is given in Section 4.8. Finally, Section 4.9 draws conclusions.

4.2 Related Work

Throughput analysis of HSDF graphs is studied extensively in [YTO91, Kar78, dKBS12, WBS07]. An algorithm proposed by Karp in [Kar78] to calculate maximum cycle mean (MCM) is an another efficient method of calculating the throughput. All these studies require a conversion of SDF graphs into HSDF graphs which can be exponentially larger than the original SDF graphs in the worst case. On the other side, the throughput calculation method applicable directly to SDF graphs [GGS$^+$06] is practical only if we only have sufficiently many processors. However, our strategy calculates maximal throughput on a given finite number of processors.

Another novel techniques for task binding and scheduling of SDF graphs under given throughput constraints are presented in [DSB$^+$11, MVB07]. These approaches use an combination of static-order scheduling and Time-Division

Multiplex (TDM). Our method, rather than utilising pre-computed static-order schedules, considers nondeterministic scheduling scheme and generates throughput-optimal schedules at design-time. Secondly, in contrast to the approach in [DSB+11], our method can handle heterogeneous systems as well. A model checking based approach to guarantee timing bounds of multiple SDF graphs running on shared-bus multi-core architectures is analysed in [FGFR13]. However, this analysis also needs a pre-computed static-order schedule.

Model checking of a recently introduced extension of SDF graphs known as Scenario-Aware Dataflow (SADF) [TGB+06] is done in [TKW12]. This approach utilises the CADP tool suite [GLMS11] by the application of Interactive Markov Chains (IMC). Nevertheless, it does not investigate the calculation of throughput or consider multiprocessor platforms.

4.3 SDF Graphs with Resource Constraints

This section first recalls the definition of SDF graphs (with time) from Chapter 2 and then introduces how we add resource constraints to SDF graphs in the form of *platform application models*.

4.3.1 SDF Graphs

An SDF graph is defined as follows.

Definition 4.1. *An SDF graph is a tuple* $G = (A, D, \mathsf{Tok}_0, \tau)$ *where:*

- A *is a finite set of actors,*
- D *is a finite set of dependency channels* $D \subseteq A^2 \times \mathbb{N}^2$*,*
- $\mathsf{Tok}_0 : D \to \mathbb{N}$ *denotes initial tokens in each channel, and*
- $\tau : A \to \mathbb{N}_{\geq 1}$ *assigns an execution time to each actor.*

A dependency channel $d = (a, b, p, q)$ *denotes a data dependency of actor* b *on actor* a*. The firing of actor* a *results in the production of* p *tokens on channel* d*. If the number of tokens on channel* d *is greater than* q*, actor* b *can execute, and as a result, it consumes* q *tokens from channel* d*.*

Example 4.2. Figure 4.2 shows an SDF graph with three actors u, v, w. Arrows between the actors depict the channels which hold tokens (dots). The execution time (ms) of the actors is represented by a number inside the actor nodes. The numbers near the source and destination of each channel are the rates. □

4.3.2 Platform Application Models

Embedded streaming applications always face tight and strict performance requirements. These applications must be processed within tight time budgets to provide customer satisfaction and good user experience. For example, the video bit rate of video compression standard H.264/AVC ranges from 64 kbps

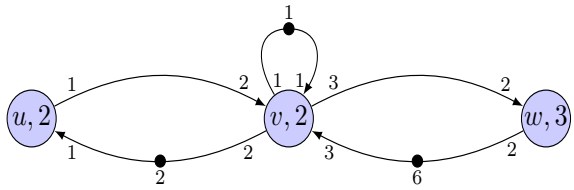


Figure 4.2: An example SDF graph

to 240 Mbps [WSBL03] depending on the user's requirements. Other than timing requirements, another factor that affects the performance is resource limitation. Having higher number of resources ensures better performance. However, due to space and cost limitations, there is always a bound on the number of resources in the system. Therefore, system designers must maintain a balance between performance requirements and resource constraints to achieve optimal solutions based on customer needs. This requires careful mapping between actors of SDF graphs and hardware components. Furthermore, in practice not all actors can be mapped onto every processor, because of memory and bandwidth limitations, analogue versus digital processing capabilities, instruction set limitations etc. Thus, the better the match between actors and underlying hardware implementations, the higher the chances that the given performance requirements are met. A good design always implies a good *mapping* between an *application* and *hardware architecture*. Therefore, we introduce *platform application models* to represent the hardware architecture as follows.

Definition 4.3. *A* platform application model *(PAM) is a tuple* $\mathcal{P} = (\Pi, \zeta)$ *consisting of*

- *a finite set of processors* $\Pi = \{\pi_1, \ldots, \pi_n\}$*, and*
- *a function* $\zeta : \Pi \to 2^A$ *indicating which actors can be mapped to which processor.*

The platform application model allows us to reason about the behaviour of an application under a specific mapping on the hardware architecture. The processor is claimed by an actor at the beginning of its firing and after the execution time of the actor elapses, it finishes firing and releases the processor as shown in Figure 4.3.

4.3.3 Example of SDF Graphs with Resource Constraints

In this subsection, we give an example of an SDF graph mapped on a PAM. Let us consider the example SDF graph shown in Figure 4.2 mapped on four processors $\Pi = \{\pi_1, \pi_2, \pi_3, \pi_4\}$. Figure 4.4 shows an example schedule of our running example. As we can see in Figure 4.4, the actor u is mapped on the processors π_1 and π_2 for its two initial firings. Afterwards, the SDF graph enters the periodic phase in which each actor is mapped on some processor to fire. As each iteration takes 12 ms, the throughput is $\frac{1}{12}$ ms^{-1}.

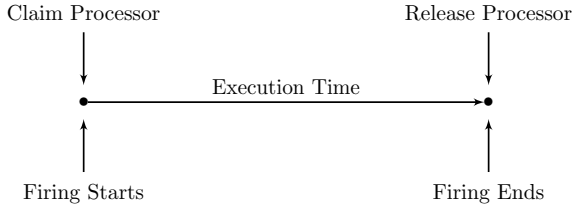


Figure 4.3: Firing of an actor (taken from [YGB+09]). An actor claims the processor at the beginning of its firing, and releases it when the firing ends.

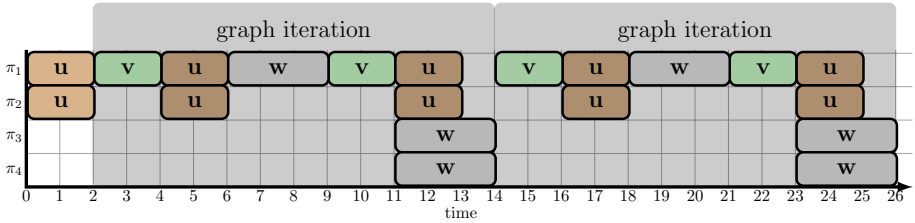


Figure 4.4: An example schedule of the SDF graph shown in Figure 4.2 on four processors π_1, π_2, π_3, and π_4 (shown on y-axis)

4.3.4 Semantics of SDF Graphs with Resource Constraints

In the following, we explain the semantics of an SDF graph G mapped on a PAM $\mathcal{P}$ in terms of a labelled transition system $\mathcal{LTS}(G_{\mathcal{P}})$. For this purpose, we define the notions of states and transitions.

Definition 4.4. *A* state *of an SDF graph* $(A, D, \mathsf{Tok}_0, \tau)$ *mapped on a platform application model (PAM)* (Π, ζ) *is a triple* $(Tok, status, TuC)$ *with the following components.*

- *channel quantity* $Tok : D \to \mathbb{N}$ *associates with each channel the number of tokens currently present in that channel, and*
- $status : \Pi \to \{idle, occup\}$ *associates with each processor* $\pi \in \Pi$ *if it is available or occupied.*
- *To observe the progress of time,* $TuC : \Pi \to \mathbb{N}$ *records for each processor the remaining execution time required to complete its current task. Here, TuC abbreviates time until completion.*

The initial state $(Tok_0, status_0, TuC_0)$ *is given by* $status_0(\pi) = idle$*, and* $TuC_0(\pi) = 0$*, for all* $\pi \in \Pi$*.*

Example 4.5. The initial state of the example explained in Section 4.3.3 is given by $(Tok_0, status_0, TuC_0) = ((0, 0, 6, 2, 1), (idle, idle, idle, idle), (0, 0, 0, 0))$. Here, initial tokens in all channels are represented by Tok_0. The initial availability of processors is given by $status_0$. Similarly, TuC_0 represents the initial remaining execution times of processors. □

Now we have defined the state of an SDF graph mapped on a PAM, we give the definition of a transition in the following.

Definition 4.6. *A* transition *of an SDF graph* $(A, D, \mathsf{Tok}_0, \tau)$ *mapped on a platform application model (PAM)* (Π, ζ) *from state* $(Tok_1, status_1, TuC_1)$ *to state* $(Tok_2, status_2, TuC_2)$ *is denoted as* $(Tok_1, status_1, TuC_1) \xrightarrow{\kappa} (Tok_2, status_2, TuC_2)$ *and label* κ *is defined as* $\kappa \in (A \times \Pi \times \{start, end\}) \cup \{tick\}$ *and corresponds to the type of transition.*

- *Label* $\kappa = (a, \pi, start)$ *denotes the starting of a firing by actor* $a \in A$ *on a processor* $\pi \in \Pi$. *This transition may occur if*
  - $Tok_1(d) \geq CR(d)$ *for all* $d \in In(a)$. *That is, all input channel* $d \in D$ *have sufficiently many tokens,*
  - $status_1(\pi) = idle$. *That is, the processor* π *is currently unoccupied, and*
  - $a \in \zeta(\pi)$. *That is, if the actor* a *can be mapped on the processor* π.

  This transition results in a new state $(Tok_2, status_2, TuC_2)$ *given by*
  - $Tok_2(d) = Tok_1(d) - CR(d)$ *for all* $d \in In(a)$. *That is, the tokens equal to the consumption rate* $CR(d)$ *are removed from each incoming channel,*
  - $status_2(\pi') = status_1(\pi')$, *and* $status_2(\pi) = occup$ *for all* $\pi' \neq \pi$. *That is, the processor* $\pi \in \Pi$ *is claimed, and*
  - $TuC_2(\pi) = \tau(a)$. *That is, the execution time of the actor* a $\tau(a)$ *is attached to the processor* π.

  Example 4.7. The actor u in the example given in Section 4.3.3 takes the transitions $(u, \pi_1, start)$ and $(u, \pi_2, start)$ at t = 0 ms. As a result, two tokens are subtracted from the channel $v - u$. □

- *Label* $\kappa = (a, \pi, end)$ *denotes the ending of a firing by actor* $a \in A$ *and releasing a processor* $\pi \in \Pi$. *This transition may occur if*
  - $status_1(\pi) = occup$. *That is, the processor* π *is currently occupied, and*
  - $TuC_1(\pi) = 0$. *That is, the actor* $a \in A$ *has finished its execution.*

  This transition results in a new state $(Tok_2, status_2, TuC_2)$ *given by*
  - $Tok_2(d) = Tok_1(d) + PR(d)$ *for all* $d \in Out(a)$. *That is, the tokens equal to the production rate* $PR(d)$ *are produced on all outgoing channels,*
  - $status_2(\pi') = status_1(\pi')$ *and* $status_2(\pi) = idle$ *for all* $\pi' \neq \pi$. *That is, the processor* $\pi \in \Pi$ *is released, and*
  - $TuC_2 = TuC_1$.

Example 4.8. In the example given in Section 4.3.3, the actor u takes the transitions (u, π_1, end) and (u, π_2, end) at t = 2 ms. As a result, two tokens are produced on the channel $u - v$. □

- *Label $\kappa = tick$ denotes a clock tick transition. This transition is enabled if,*
  - $\forall \pi' \in \Pi$, $TuC_1(\pi') \neq 0$. *That is, no end transition is enabled.*

  For all $d' \in D$ and $\pi' \in \Pi$, this transition results in a new state (Tok_2, $status_2$, TuC_2) given by
  - $Tok_2(d') = Tok_1(d')$,
  - $status_2(\pi') = status_1(\pi')$, *and*
  - $TuC_2(\pi') = TuC_1(\pi') - 1$. *That is, the remaining execution time assigned to the processors is decreased by 1.*

Example 4.9. In our running example, at t = 13 ms, the actor u takes the transitions (u, π_1, end) and (u, π_2, end). Then, between t = 13 and 14 ms, there is no enabled *end* transition, and the next *end* transitions (w, π_3, end) and (w, π_4, end) are at t = 14 ms taken by the actor w. Thus, there is one *tick* transition between t = 13 and 14 ms. □

By defining LTS $\mathcal{LTS}(G_{\mathcal{P}})$ of an SDF graph G when mapped on a PAM $\mathcal{P}$ in terms of states and transitions, it is easier to understand the underlying semantics of SDF graphs with resource constraints. In the following, we describe the throughput analysis of an SDF graph G with resource constraints with the help of its LTS $\mathcal{LTS}(G_{\mathcal{P}})$.

4.4 Throughput Analysis of SDF Graphs with Resource Constraints

This section illustrates the throughput analysis of an SDF graph with resource constraints. Recall from Chapter 2 that the maximal throughput of an SDF graph *without* resource constraints is determined from *self-timed execution* [GGS+06] in which every actor fires *as soon as* it is enabled. Similar to self-timed execution, we will show that the state-space of an SDF graph with resource constraints also contains a transient phase followed by a periodic phase. As we have seen in Section 2.2, the notion of resources is not relevant for detecting the periodic phase, and calculating throughput. Thus, we consider the definition of the state (ρ, υ) given in Section 2.2.

Let (ρ_0, υ_0) and (ρ_r, υ_r) denote the initial and recurrent states at the completion of the periodic phase respectively in a self-timed execution. For each actor $a \in A$, let $\mathcal{F}_{a_t}$ and $\mathcal{F}_{a_p}$ represents the number of times actor $a \in A$ fires in the transient and periodic phase respectively. We also define the number of iterations per period as *iter*.

Lemma 4.10. *If a periodic phase in a self-timed execution is repeated for n' times, then $\mathcal{F}_{a_p}$ is equal to $n' \cdot iter \cdot \gamma(a)$.*

Proof. The proof follows from the definition of self-timed execution, repetition vector, and iteration in Chapter 2. □

The self-timed execution takes the minimum amount of time to revisit (ρ_r, υ_r) and provides the maximum throughput of an SDF graph. Therefore, we can consider it as a *fastest execution* to reach (ρ_r, υ_r) again.

As a result of the fastest execution, let us say that the SDF graph has repeated the periodic phase n' times and is in the state (ρ_r, υ_r). Then, we can say the following.

Lemma 4.11. *The SDF graph G reaches the initial state (ρ_0, υ_0) if G is executed from the state (ρ_r, υ_r) in such a way that each actor $a \in A$ fires equal to $\mathcal{F}'_{a_t} = k \cdot \gamma(a) - \mathcal{F}_{a_t}$ for some constant k.*

Proof. Total number of firings for each actor $a \in A$ in this case are:

$$= \mathcal{F}_{a_t} + \mathcal{F}_{a_p} + \mathcal{F}'_{a_t}$$

$$= \mathcal{F}_{a_t} + n' \cdot iter \cdot \gamma(a) + k \cdot \gamma(a) - \mathcal{F}_{a_t}$$

$$= (n' \cdot iter + k) \cdot \gamma(a)$$

From Fact 2.25, $\Gamma(n' \cdot \gamma) = 0$ for any constant n'. □

A necessary condition for previous lemma to hold is $\mathcal{F}'_{a_t} \geq 0$ by having a suitable value for k. If the value of k is not large enough, $\mathcal{F}'_{a_t}$ can be less than 0, which will violate Lemma 4.11. To reach (ρ_0, υ_0) from (ρ_r, υ_r) in the least number of firings, $\mathcal{F}'_{a_t}$ must be minimal. Let k_{min} denotes the smallest k such that $\mathcal{F}'_{a_t} \geq 0$ and $\mathcal{F}'_{a_t}$ is minimal for all actors $a \in A$.

If we assume that the part of execution from (ρ_r, υ_r) to (ρ_0, υ_0) is *fastest* also, then we can say the following.

Lemma 4.12. *The fastest execution of every consistent and strongly connected SDF graph repeats the periodic phase n' times if each actor $a \in A$ fires equal to $(n' \cdot iter + k_{min}) \cdot \gamma(a)$ for some constants n' and k_{min}.*

Proof. Trivial following Lemma 4.11. If a transient phase does not exist and the SDF graph enters the periodic phase directly, then $\mathcal{F}_{a_t} = 0$. In this case, the minimum value of k satisfying $\mathcal{F}'_{a_t} \geq 0$ is $k_{min} = 0$. Furthermore, the total number of firings is equal to $n' \cdot iter \cdot \gamma(a)$ for each $a \in A$ and the periodic phase is repeated n' times. □

We propose UPPAAL as a tool to compute the repetition vector and throughput. UPPAAL can automatically verify a number of properties, including invariant and reachability checking. An important feature in our approach is the option of generating a trace with the shortest possible accumulated time delay to reach the final state, i.e., $(n' \cdot iter + k_{min}) \cdot \gamma(a)$ for each actor $a \in A$ from the initial state (ρ_0, υ_0), termed *Fastest Trace*. UPPAAL explores the whole state-space and finds the fastest execution trace containing the periodic phase repeated n' times. From the periodic phase, we determine the maximal throughput of the SDF graph.

Self-timed execution assumes that there is an *unbounded* number of processors to accommodate all enabled firings of all actors at a certain time. Let Π^{min} denotes a finite set containing the minimum number of processors required to allow self-timed execution. From Lemmas 4.11 and 4.12, we can generalise the following.

Lemma 4.13. *For every consistent and strongly connected SDF graph mapped on a platform application model (PAM)* (Π, ζ) *in such a way that* $\bigcup \zeta(\pi) = A$ *and* $\emptyset \subset \Pi \subseteq \Pi^{min}$*, the maximal throughput of an SDF graph is determined from the periodic phase of the fastest execution to the* i^{th} *multiple of the repetition vector, for some constant* i*.*

Proof. In a strongly connected and consistent SDF graph, each actor depends on other actors in order to have a sufficient amount of tokens on its input channels to be enabled for firing. This implies a bound on the difference in the number of firings of each actor with respect to the corresponding entries in the repetition vector. The state-space of reaching the i^{th} multiple of the repetition vector for some constant i if $\emptyset \subset \Pi \subseteq \Pi^{min}$ could contain multiple possible executions. If we search the whole state-space and consider only the fastest execution out of all executions, we notice that it contains a periodic phase implying the maximal throughput.

The reason is that in a fastest execution, if insufficient processors are available to map all simultaneous enabled firings, some of the firings will be delayed. Delaying a certain firing does not change any dependency. Instead, successors firings would also be delayed. The constraint of having to reach the final state in the least possible time ensures that delayed firings are mapped in such a way that they cause the least delay for their successor firings to be enabled. As the number of simultaneous firings of the actors and number of tokens in any channel remains bounded, the state-space is also finite. This ensures that a certain state (ρ_r, υ_r) will be revisited eventually during the execution representing the periodic phase. We explore the whole state-space with UPPAAL and find the fastest execution trace from all possible executions. □

For each SDF graph, the value of k_{min} varies by altering the given number of processors and depends on how many times each actor $a \in A$ has fired during the transient phase. Therefore, the value of $n' \cdot iter + k_{min}$ given to UPPAAL as a final state must be large enough to ensure that $\mathcal{F}'_{a_t}$ is greater than 0 and the SDF graph enters the periodic phase.

Example 4.14. Figure 4.5 shows the self-timed execution for the SDF graph in Figure 4.2 on page 73. If we compare it with the example schedule in Figure 4.4 on page 74, we see that each actor is firing as soon as it enabled in Figure 4.5. For example, the actor w is enabled after the first firing of the actor v. Therefore, in the self-timed execution in Figure 4.5, the actor w fires immediately after the first firing of the actor v is finished. Whereas, in the example schedule in Figure 4.4, the actor w has to wait 2 ms extra to fire.

As we can further see in Figure 4.5, the minimum number of processors to achieve the self-timed execution is 4, i.e., $|\Pi^{min}| = 4$. If we map the same SDF

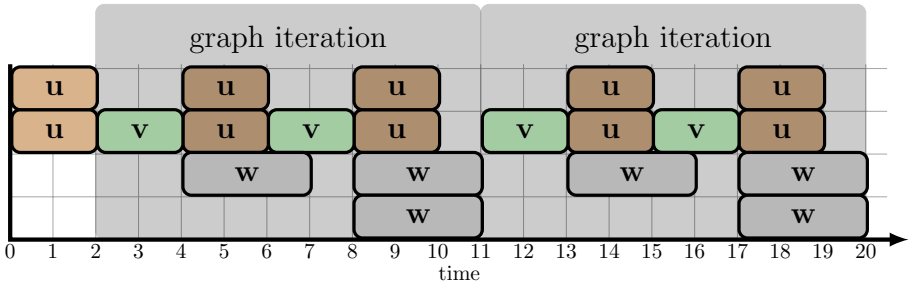


Figure 4.5: Self-timed execution of the SDF graph shown in Figure 4.2. After the transient phase (initial two firings of u), the SDF graph enters the periodic phase shown by the shaded regions.

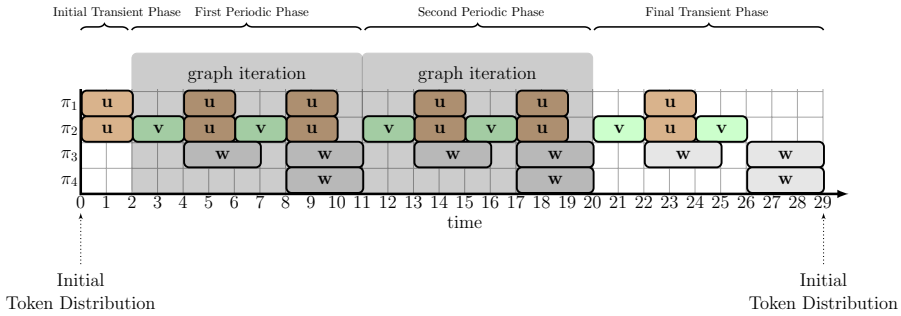


Figure 4.6: Schedule by fastest execution to 3^{rd} multiple of repetition vector on four processors π_1, π_2, π_3 and π_4. After the initial transient phase (initial two firings of u), the SDF graph enters the periodic phase shown by the shaded regions. The SDF graph repeats the periodic phase twice, and then enters the final transient phase.

graph on four processors $\{\pi_1, \pi_2, \pi_3, \pi_4\}$, then the fastest execution to the 3^{rd} multiple of repetition vector, i.e., $3{\cdot}\gamma = \langle 12, 6, 9\rangle$ is shown in Figure 4.6. In this example, the values of n', *iter* and k_{min} are 2, 1 and 1 respectively. Therefore, the periodic phase is repeated twice. We could determine the throughput from the periodic phase which is equal to $\frac{1}{9}$ ms^{-1}.

We can also observe that the firing sequence of the actors in the self-timed execution is same as fastest execution, except the final transient phase. In the rest of chapter, we do not analyse the final transient phase as it does not affect throughput.

4.5 From SDF Graphs and Platform Application Models to Timed Automata

This section first explains the translation of SDF graphs and platform application models (PAMs) to timed automata (TA), and then implementation of this translation in UPPAAL.

4.5.1 Translation of SDF Graphs and PAMs to Timed Automata

Our framework of scheduling SDF graphs consists of separate models of an SDF graph and the processors in a PAM. This method splits the scheduling problem of the SDF graphs in terms of the tasks and resources.

Given an SDF graph $G = (A, D, \mathsf{Tok}_0, \tau)$ together with a platform application model (PAM) $\mathcal{P} = (\Pi, \zeta)$, we generate a parallel composition of timed automata (TA):

$$A_G \| Processor_1 \| \ldots \| Processor_n,$$

as shown in Figure 4.7 on page 84. Here, the timed automaton A_G models the SDF graph as shown in Figure 4.7a. The TA $Processor_1, \ldots, Processor_n$ model the processors $\Pi = \{\pi_1, \ldots, \pi_n\}$, as shown in Figure 4.7b. The underlying LTS $\mathcal{LTS}(G_\mathcal{P})$ of G mapped on $\mathcal{P}$ is given by $(S, Lab, \rightarrow_G)$ where $S = (Tok, status, TuC)$ denotes the states, $Lab = \kappa$ denotes the labels, and $\rightarrow_G \subseteq S \times Lab \times S$ depicts the transitions.

Timed Automaton A_G. The timed automaton A_G models the SDF graph G. Given G and $\mathcal{P}$, the automaton A_G is defined as

$$A_G = (L, l^0, Act, C, E, Inv)$$

All components of A_G are explained as follows.

- The location $L = l_0 = \{\mathsf{Initial}\}$ is the only location in our SDF graph model.
- The action set $Act = \{\mathsf{fire!}, \mathsf{end?}\}$ contains two parametrised actions to synchronise with the TA $Processor_1, \ldots, Processor_n$. The first action $\mathsf{fire}[\pi][\mathsf{a}]!$ (exclamation mark signifies a sending operation) represents the start of the firing of an actor a on a processor π for each $\pi \in \Pi$ and $a \in A$. The second action $\mathsf{end}[\pi][\mathsf{a}]?$ (question mark signifies a receiving operation) represents the end of the firing of an actor a on a processor π for each $\pi \in \Pi$ and $a \in A$.
- The automaton A_G does not contain any clocks and invariants. Therefore, *Inv:* $L \rightarrow B(C)$ and $Inv(l^0) = true$.
- The action $\mathsf{fire}[\pi][\mathsf{a}]$ is enabled if the incoming channels of $a \in A$ have sufficient tokens. For each $a \in A$ and all $d \in In(a)$, the edge set E contains two edges such that:
  - $\mathsf{Initial} \xrightarrow{Tok(d) \geq CR(d) : \mathsf{fire}[\pi][\mathsf{a}]!,\ Tok(d) = Tok(d) - CR(d)} \mathsf{Initial}$

– $\mathsf{Initial} \xrightarrow{true:\mathsf{end}[\pi][\mathsf{a}]?,\, Tok(d)=Tok(d)+PR(d)} \mathsf{Initial}$.

Here, $Tok(d) \geq CR(d)$ refers to a *guard* and it signifies that tokens on all input channels $d \in In(a)$ of an actor $a \in A$ must be greater than or equal to their consumption rate in order to take the action fire!. As a result of taking the action fire!, tokens on all input channels $d \in In(a)$ of an actor $a \in A$ are subtracted, i.e., $Tok(d) = Tok(d) - CR(d)$. Similarly, by taking the action end?, the actor firing is completed and tokens are produced on all output channels $d \in Out(a)$ of an actor $a \in A$, i.e., $Tok(d) = Tok(d) + PR(d)$.

The automaton A_G contains a number of variables: for each channel from actors $a \in A$ to $b \in B$, an integer variable $\mathsf{buff\_a2b} = Tok(\mathsf{a}, \mathsf{b}, \mathsf{p}, \mathsf{q})$ contains the number of tokens in the channel from a to b. The boolean variable flag_active which is initially *false*, is set to *true* as soon as any actor fires. The variable counter_a counts how many times the actor $a \in A$ has been fired. Initially, $\mathsf{counter\_a} = 0$ and $\mathsf{buff\_a2b} = Tok_0(\mathsf{a}, \mathsf{b}, \mathsf{p}, \mathsf{q})$ contains the number of tokens in the initial distribution of the channel (a, b, p, q).

Taking the action $\mathsf{fire}[\pi][\mathsf{a}]$ consumes, for each actor $a \in A$ and input channel $(b, a, p, q) \in In(a)$ in G, the q tokens from the channel buff_b2a, and is carried out by the function $\mathsf{consume}(\mathsf{buff\_b2a}, \mathsf{q})$. The action $\mathsf{end}[\pi][\mathsf{a}]$ adds, for each actor $a \in A$ and output channel $(a, b, p, q) \in Out(a)$ in G, the p tokens on the buffer buff_a2b by carrying out the function $\mathsf{produce}(\mathsf{buff\_a2b}, \mathsf{p})$.

Timed Automata Processor$_j$. The timed automata *Processor$_j$* model PAMs. For each $\pi_j \in \Pi$, we define the processor TA

$$Processor_j = (L_j, l_j^0, Act_j, C_j, E_j, Inv_j).$$

All components of *Processor$_j$* are explained as follows.

- The TA *Processor$_j$* include both an idle and occupied state. That is, for each $a \in \zeta(\pi_j)$, let $L_j = \{\mathsf{Idle}, \mathsf{InUse\_a} | \mathsf{a} \in \mathsf{A}\}$ indicating that the processor $\pi_j \in \Pi$ is either in the idle state, or is currently occupied by the actor $a \in A$.

- The initial location is given by $l_j^0 = \mathsf{Idle}$. This represents that initially, each processor is in the idle state.

- The action set $Act = \{\mathsf{fire?}, \mathsf{end!}\}$ contains two parametrised actions to synchronise with the timed automaton A_G. The first action $\mathsf{fire}[\pi][\mathsf{a}]?$ represents the start of the firing of an actor a on a processor $\pi \in \Pi$. This action further signifies that the processor π is currently in use. The second action $\mathsf{end}[\pi][\mathsf{a}]!$ represents the end of the firing of an actor a on a processor $\pi \in \Pi$. This action also signifies the releasing of the processor π.

- The TA *Processor$_j$* has only clock $C_j = \{x_j\}$. Since clocks in UPPAAL are local, we can abbreviate x_j by x.

- For each $\pi_j \in \Pi$ and $a \in \zeta(\pi_j)$, the edge set E contains two edges,

  - $\mathsf{Idle} \xrightarrow{true:\mathsf{fire}[\pi][\mathsf{a}]?, x=0} \mathsf{InUse\_a}$, and
  - $\mathsf{InUse\_a} \xrightarrow{x=\tau(a):\mathsf{end}[\pi][\mathsf{a}]!, \varnothing} \mathsf{Idle}$.

- Each location InUse_a is equipped with an invariant $Inv_j(\mathsf{InUse\_a}) \leq \tau(\mathsf{a})$ enforcing the system to stay in InUse_a for less than or equal to the execution time $\tau(a)$.

The action $\mathsf{fire}[\pi][\mathsf{a}]$ is enabled in the idle state Idle and leads to the location InUse_a. Thus, $\mathsf{fire}[\pi][\mathsf{a}]$ "claims" the processor $\pi \in \Pi$, so that any other firing cannot run on $\pi \in \Pi$ before the current firing of $a \in A$ is finished. As each location InUse_a has an invariant $Inv_j(\mathsf{InUse\_a}) \leq \tau(\mathsf{a})$, the automaton can stay in InUse_a for at most until the execution time $\tau(a)$. If $x = \tau(a)$, the system has to leave InUse_a by taking the action $\mathsf{end}[\pi][\mathsf{a}]$. In this way, A_G is notified that the execution of $a \in A$ has ended, so that A_G updates the SDF graph channels and other variables.

In the next subsection, we will describe the implementation of the translation explained above in UPPAAL. For this purpose, we take help of the example SDF graph in Figure 4.2 on page 73.

4.5.2 Modelling SDF Graphs and PAMs in UPPAAL

Let us consider an SDF graph in Figure 4.2 and its self-timed execution shown in Figure 4.5 on page 79. Following the setup in $A_G \| Processor_1 \| \ldots \| Processor_n$, we build a separate template for the SDF graph and PAM namely SDFG and Processor respectively in UPPAAL. As we need four processors to observe the self-timed execution, we create four instances of the Processor template. Each actor in SDFG and each instantiation of the Processor template is given an unique id and passed as parameters to the templates. Whole system is comprised of one instance of SDFG called SDF_Graph and four instances of Processor called Processor1, Processor2, Processor3 and Processor4 as it is declared in Listing 4.1.

Figure 4.7 on page 84 shows the models of SDFG and Processor in the editor of UPPAAL and Listing 4.2 shows all global declarations used in these templates. There are two channels for each actor and a single location Initial. The parameters consist of ids of each actor. The label e : id_r selects processor ids from the user-defined type id_r declared in Listing 4.2 by which the SDF graph template SDFG communicates with the processor template Processor. For each channel in an SDF graph, there is an integer variable in the UPPAAL model. The initial value of this variable is equal to the initial number of tokens in the channel. For example, in Listing 4.2, the initial tokens in the channel from actor $w \in A$ to actor $v \in A$ are defined by int buff_w2v = 6;. The constant variables N and M denote the given number of processors, and the actors respectively. The channels fire[N][M] and end[N][M] are used to synchronise both templates. The functions produce (consume) respectively produces (consumes) tokens equal to production (consumption) rate of the particular channel. The integer variables counter_u, counter_v and counter_w count the number of times actor u, v and w fires respectively. The boolean variable flag_act has an initial value equal to *false* and its value changes to *true* as soon as any actor fires. In Listing 4.2, the

Listing 4.1: System declarations in UPPAAL for system in Section 4.5.2

```
1  // Actor ids
2  const int u=0;
3  const int v=1;
4  const int w=2;
5
6  // Processor ids
7  const int p1=1;
8  const int p2=2;
9  const int p3=3;
10 const int p4=4;
11
12 // SDF Graph template instantiation
13 SDF_Graph = SDFG(u,v,w);
14
15 // Processor template instantiation
16 Processor1 = Processor(p1,u,v,w);
17 Processor2 = Processor(p2,u,v,w);
18 Processor3 = Processor(p3,u,v,w);
19 Processor4 = Processor(p4,u,v,w);
20
21 // Processes to be composed into a system.
22 system SDF_Graph, Processor1, Processor2, Processor3, Processor4
       ;
```

clock variable global observes the overall time progress of any trace. The clock variable x of the processor is declared as a local variable (not shown here).

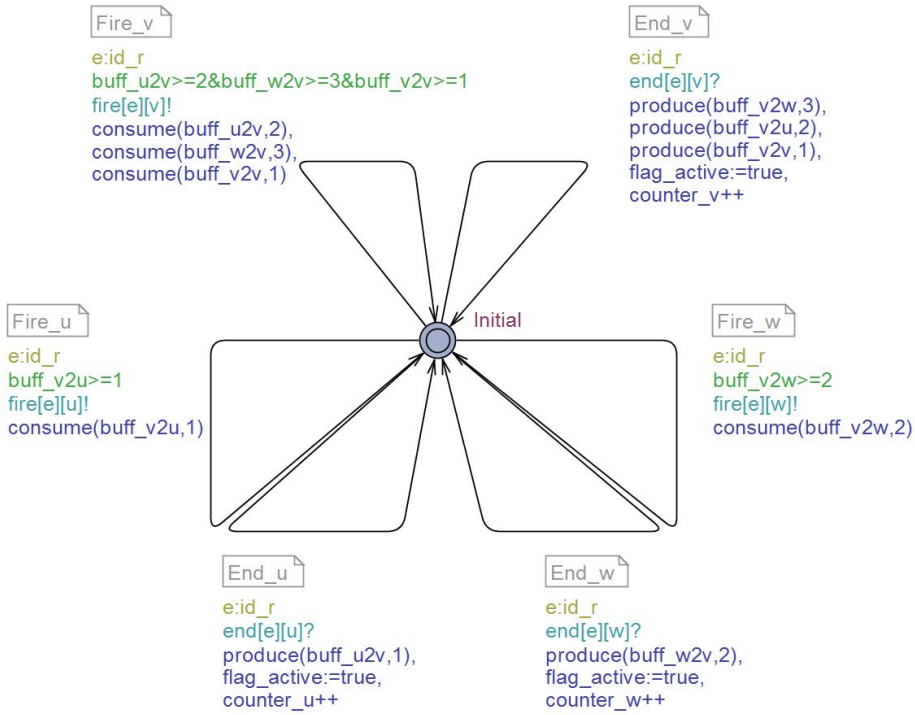
The location Idle in the Processor model in Figure 4.7b is an initial location and InUse_u, InUse_v and InUse_w are the dedicated locations for each actor. In this model, the processor ids are represented by p_id and are passed as parameters.

4.6 Resource-Constrained Scheduling of SDF Graphs using UPPAAL

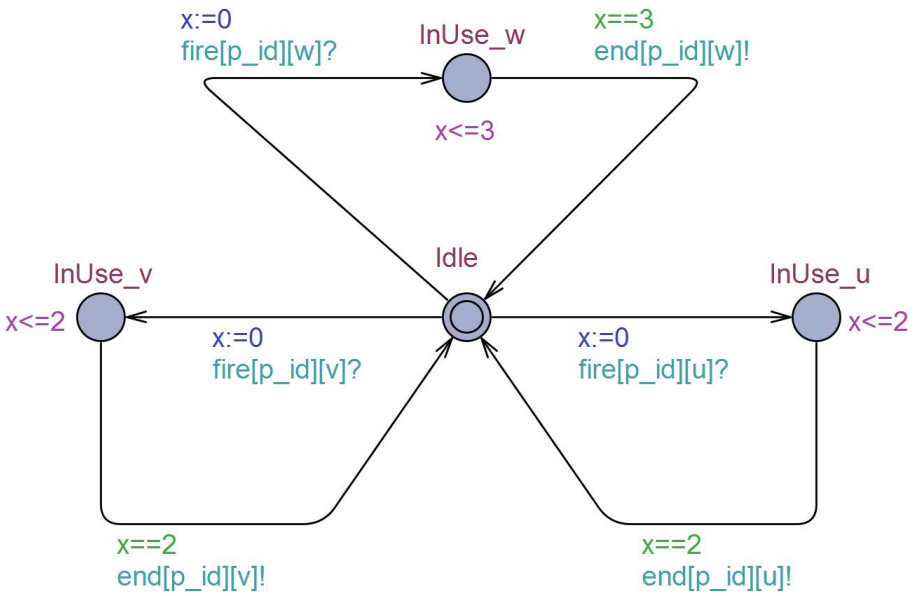
In this section, we will describe scheduling of SDF graphs on a given number of processors using UPPAAL. For this purpose, we consider the SDF graph in Figure 4.2 on page 73 as an example.

4.6.1 Throughput Calculation

Following Lemma 4.13, starting from the initial token distribution of an SDF graph, we ask UPPAAL to find a trace which leads us to the initial token distribution again in the least possible time. We have a boolean variable flag_active with an initial value false in our UPPAAL model. As soon as the UPPAAL model starts executing, the value of flag_active changes to true. In a nutshell, the purpose of flag_active is not to give the initial state as a result and to force the model to start executing. We also associate a counter with each actor. By checking the



(a) UPPAAL model A_G for SDF graph in Figure 4.2 having three actors u, v, w



(b) UPPAAL model *Processor*$_j$ onto which SDF graph in Figure 4.2 is mapped

Figure 4.7: UPPAAL editor showing SDF graph and Processor TA models

Listing 4.2: Global declarations in UPPAAL for SDF graph in Figure 4.2

```
1  //Global Clock
2  clock global;
3
4  const int N = 4;     //# of Processors
5  const int M = 3;     //# of Actors
6
7  //Processors IDs
8  typedef int[1,N] id_r;
9
10 //UPPAAL Channels
11 chan end[N][M], fire[N][M];
12
13 //SDF graph channel sizes
14 int buff_u2v,buff_v2w=0;
15 int buff_v2u=2;
16 int buff_w2v=6;
17 int buff_v2v=1;
18
19 //Flag to check if SDF graph has started
20 bool flag_active=false;
21
22 //Counter for each actor
23 int counter_u, counter_v, counter_w=0;
24
25 //Consume function for consuming tokens from incoming channels
26 void consume(int &channel_tokens, int tokens)
27 {
28         channel_tokens-=tokens;
29 }
30
31 //Produce function for producing tokens on outgoing channels
32 void produce(int &channel_tokens, int tokens)
33 {
34      channel_tokens+=tokens;
35 }
```

values of counters, we determine how many times each actor has fired to reach the target state (initial token distribution) which gives us the *repetition vector*.

As we know the initial token distribution of the SDF graph in Figure 4.2, selecting Fastest trace and verifying the following query in UPPAAL generates a trace by which we determine the repetition vector.

$$\mathbf{E} <> (\texttt{buff\_u2v} == 0 \texttt{ and } \texttt{buff\_v2w} == 0 \texttt{ and } \texttt{buff\_v2u} == 2 \texttt{ and } \texttt{buff\_w2v} == 6 \texttt{ and } \texttt{buff\_v2v} == 1 \texttt{ and } \texttt{flag\_active} == \texttt{true})$$

As a result of this query, a trace is generated and by examining the variables counter_u, counter_v and counter_w, we can determine the value of repetition vector, i.e., $\langle$u,v,w$\rangle$=$\langle 4, 2, 3\rangle$.

The repetition vector γ found in the previous step is an input to find

the *maximal throughput*. Following Lemma 4.13, we find the fastest trace to $n' \cdot iter + k_{min}$-multiple of the repetition vector.

We find out the throughput of the SDF graph shown in Figure 4.2 on page 73 using $n' \cdot iter + k_{min} = 3^{rd}$ multiple of the repetition vector, i.e., $\langle 12, 6, 9 \rangle$ by verifying the following query.

$$\mathbf{E} <> (\texttt{counter\_u} == 12 \texttt{ and } \texttt{counter\_v} == 6 \texttt{ and } \texttt{counter\_w} == 9)$$

Figure 4.6 on page 79 shows the schedule build from the generated trace when the SDF graph in Figure 4.2 on page 73 is mapped on four processors.

Similarly, we can detect the presence or absence of *deadlocks* in an SDF graph by checking "**A**[] not deadlock".

Using the results presented earlier, if we model the same SDF graph with three processors in UPPAAL, we get a schedule shown in Figure 4.8. We can observe that even we have reduced the number of processors from four to three, the throughput still is $\frac{1}{9}$ ms^{-1} which clearly shows that we do not always need a self-timed execution to realise the maximum throughput.

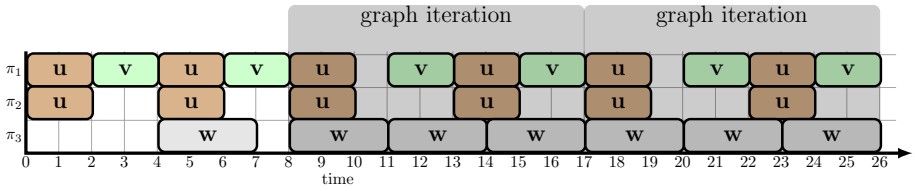


Figure 4.8: Schedule by fastest execution on three processors π_1, π_2 and π_3.

In the same fashion, Figure 4.9 shows a schedule using two processors. Thus, even we have reduced the number of processors by one, the throughput does not deteriorate significantly and decreases slightly to $\frac{1}{11}$ ms^{-1}. The Pareto space in terms of throughput and number of processors is shown in Figure 4.10.

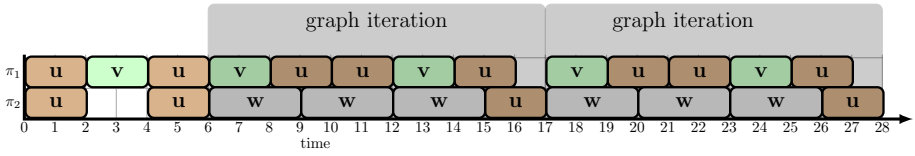


Figure 4.9: Schedule by fastest execution on two processors π_1 and π_2

Table 4.11 shows the results for peak memory consumption and computation time needed to find out throughput and deadlock freedom for the SDF graph shown in Figure 4.2 on page 73. These figures are determined using an utility called *memtime* [MEM02]. The experiments were run on a dual-core 2.8 GHz machine with 4 GB RAM. The first column displays the number of processors, and the second column represents the value of *maximal* throughput (ms^{-1}) with respect to various numbers of processors. Columns 3-6 depict the memory consumption (KB) and computation time (s) required by UPPAAL in generating

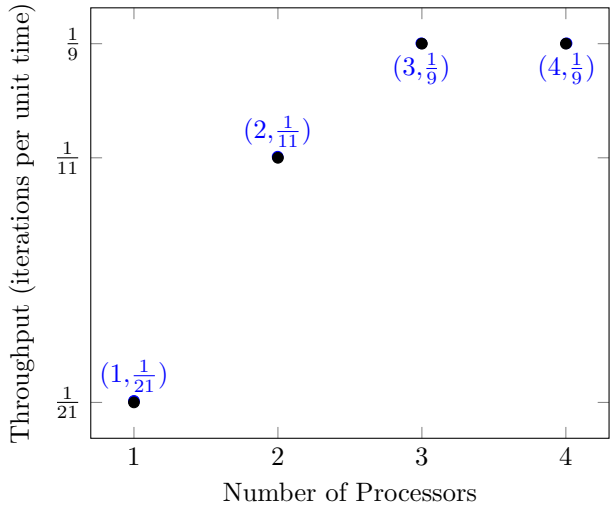


Figure 4.10: Pareto space showing number of processors and maximum throughput for SDF graph in Figure 4.2

Number of Processors	Maximal Throughput	Max. Throughput		Deadlock Freedom		SDF^3
		Memory	Time	Memory	Time	Time
4 (self-timed)	1/9	38144	0.3	37880	0.21	0
3	1/9	2008	0.1	2008	0.1	-
2	1/11	2008	0.1	2008	0.1	-
1	1/21	2008	0.1	2008	0.1	-

Table 4.11: Experimental Results for SDF graph in Figure 4.2 regarding throughput with respect to varying numbers of processors

the *fastest* trace of the second multiple of the repetition vector to determine throughput, and to verify deadlock freedom. The final column represents time (s) taken by SDF^3 for calculating the throughput for self-timed execution. It also explains that SDF^3 only calculates the throughput of an SDF graph assuming that sufficient number of processors to realise self-timed execution are available.

4.6.2 Scheduling in a Heterogeneous System

So far, we have assumed a homogeneous system in which an actor can be mapped on any processor as all processors are identical. A homogeneous system gives more freedom to decide which actor to assign to a particular processor. However, this freedom is constrained in a heterogeneous system by which processors could be utilised to execute a particular actor.

In UPPAAL, we can utilise the same models described earlier in a heterogeneous system following Lemma 4.13. Let us consider the SDF graph in Figure 4.2 on page 73 mapped on a heterogeneous system in such a way that the actor u can

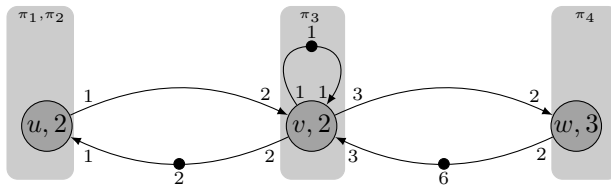


Figure 4.12: SDF graph in Figure 4.2 mapped on a heterogeneous system. Actor u is mapped on the processors π_1 and π_2, actor v is mapped on the processor π_3, and the processor π_4 is assigned to execute actor w.

Listing 4.3: System declarations in UPPAAL for Heterogeneous System in Section 4.6.2

```
1  // Actor ids
2  const int u=0;
3  const int v=1;
4  const int w=2;
5  const int dummy=3;
6
7  // Processor ids
8  const int p1=1;
9  const int p2=2;
10 const int p3=3;
11 const int p4=4;
12
13 // SDF Graph template instantiation
14 SDF_Graph = SDFG(u,v,w);
15
16 // Processor template instantiation
17 Processor1 = Processor(p1,u,dummy,dummy);
18 Processor2 = Processor(p2,u,dummy,dummy);
19 Processor3 = Processor(p3,dummy,v,dummy);
20 Processor4 = Processor(p4,dummy,dummy,w);
21
22 // Processes to be composed into a system.
23 system SDF_Graph, Processor1, Processor2, Processor3, Processor4
       ;
```

be mapped only on the processors π_1 and π_2, the actor v can be executed only on the processor π_3, and the processor π_4 is assigned to execute the actor w only, as shown in Figure 4.12. This setting is shown in Listing 4.3.

We change the value of variable M to four in Listing 4.2 and introduce a dummy actor in "System declarations" as mentioned in Listing 4.3. We can see in Listing 4.3 that the dummy actor is passed as a parameter in place of those actors which are not to be bound to a particular processor The schedule of this heterogeneous system is shown in Figure 4.13 and the *maximal* throughput achieved is $\frac{1}{9}$ ms^{-1}.

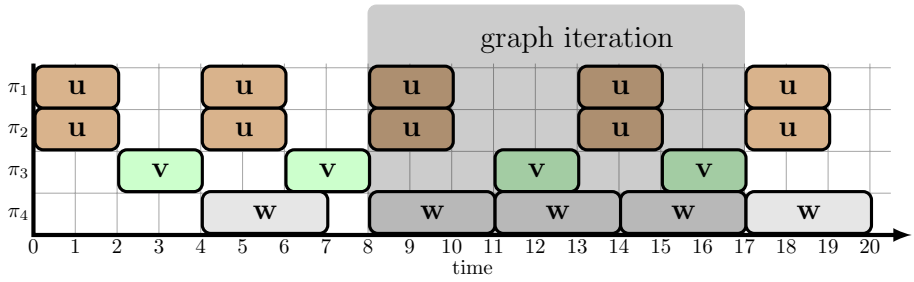


Figure 4.13: Schedule of the heterogeneous system in Figure 4.12 on four processors π_1, π_2, π_3 and π_4

4.7 Case Studies

This section presents the results of the experiments in various case studies given in Chapter 2 namely, an MPEG-4 decoder in Figure 2.14, an MP3 decoder in Figure 2.15, an MP3 playback application in Figure 2.16, an audio echo canceller in Figure 2.17, and a bipartite graph with buffer capacities in Figure 2.18. We also consider an example SDF graph given in Figure 4.14.

Table 4.15 records the repetition vector of each SDF graph and Table 4.16 displays the results of the experiments of generating the *fastest* trace of the second multiple of the repetition vector to find out throughput (ms^{-1}), verify deadlock freedom and comparison with SDF^3.

We could determine the exact number of processors required for a self-timed execution, using SDF^3. Then, we apply our approach to derive an optimal schedule

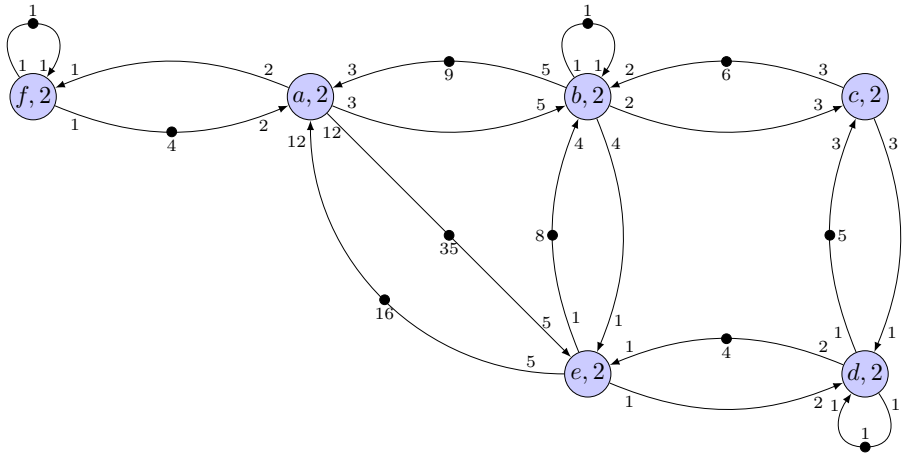


Figure 4.14: An example SDF graph case study

Case Studies	Repetition Vector
MPEG-4 Decoder in Figure 2.14	[FD VLD IDCT RC MC] = [1 5 5 1 1]
MP3 Decoder in Figure 2.15	[Huffman, Req0, Req1, Redorder0, Reorder1, Stereo, Antialias0, Antialias1, Hyb Syn.0, Hyb Syn.1, Freq. Inv0, Freq. Inv1, Subb. Inv0, Subb. Inv1] = [2 1 1 1 1 1 1 1 1 1 1 1 1]
MP3 Playback Application in Figure 2.16	[MP3 SRC DAC] = [3 235 1880]
Audio Echo Canceller in Figure 2.17	[OUT SRC AEC ADC] = [23 23 1 23]
Bipartite graph in Figure 2.18	[a b c d] = [12 36 9 16]
Example SDF graph in Figure 4.14	[a b c d e f] = [5 3 2 6 12 10]

Table 4.15: Repetition vectors of case studies calculated using UPPAAL

on fewer processors. Thus, using model checking, we could generate an optimal schedule in a simple manner on a given number of processors automatically, once the target state is specified in a query.

4.8 Tool Support

In previous sections, we proposed UPPAAL for optimal scheduling of SDF graphs. We explained translation of SDF graphs to timed automata in Section 4.5.1, UPPAAL models in Section 4.5.2, and utilising UPPAAL for deriving optimal schedules in Section 4.6. To automate these steps, we developed a tool-chain based on Eclipse Modeling Framework (EMF) [SBMP08] termed STARS (**S**cheduling and **T**emporal **A**nalysis on limited **R**esources for **S**DF graphs). It takes as an input an SDF graph generated using the well-known tool termed SDF^3, and a PAM. Then, it transforms these components to UPPAAL, which in turn outputs the optimal schedules. Figure 4.17 on page 92 shows the workflow of the STARS tool-chain.

4.8.1 Input

The STARS tool-chain takes as an input SDF graphs generated using SDF^3 in XML format. However, SDF^3 only provides throughput of a self-timed execution where each actor is fired as soon as it is enabled. Thus, it is assumed that sufficient processors are available to accommodate all enabled firings simultaneously. Nevertheless, SDF^3 provides a limited support for specifying hardware architecture in the form of *processor types* and execution time of actors on these *processor types*. We use this feature to develop PAMs.

All components of the STARS tool-chain can be found at `https://github.com/utwente-fmt/STARS`. An instruction manual to use the STARS tool-chain is also given in this repository.

Number of Processors	Maximal Throughput	Max. Throughput		Deadlock Freedom		SDF^3
		Memory	Time	Memory	Time	Time
MPEG-4 Decoder in Figure 2.14						
6 (self-timed)	1/4	99460	259.18	41576	3.5	0
5	1/5	48960	12.04	39320	1.11	-
4	1/5	39628	0.71	38268	0.41	-
3	1/6	2008	0.11	38008	0.2	-
2	1/8	2008	0.1	2008	0.11	-
1	1/13	2008	0.1	2008	0.1	-
MP3 Decoder in Figure 2.15						
2 (self-timed)	1/9	38172	0.22	2008	0.1	0
1	1/15	2008	0.1	2008	0.1	-
MP3 Playback Application in Figure 2.16						
2 (self-timed)	1/1880	99176	7.25	67056	8.93	0.036002
1	1/2118	59472	1.41	47248	2.1	-
Audio Echo Canceller in Figure 2.17						
4 (self-timed)	1/23	2874728	302.97	1820852	856.36	0.004
3	1/24	484736	133.65	578080	181.36	-
2	1/25	149264	18.29	150088	26.46	-
1	1/70	55572	1.41	60856	2.82	-
Bipartite graph in Figure 2.18						
4 (self-timed)	1/42	38036	0.41	38024	0.21	0
3	1/44	37880	0.31	38008	0.2	-
2	1/51	37884	0.21	2008	0.1	-
1	1/73	2008	0.1	2008	0.1	-
Example SDF graph in Figure 4.14						
5 (self-timed)	1/24	153048	108.48	71932	36.2	0
4	1/24	63924	10.28	48600	0.2	-
3	1/28	2008	0.1	40500	1.92	-
2	1/38	2008	0.1	38284	0.3	-
1	1/76	2008	0.1	2008	0.1	-

Table 4.16: Maximum throughput of case studies calculated using UPPAAL, and comparison with SDF^3

4.8.2 Transforming SDF^3 Models to UPPAAL Models

After creating SDF^3 models, the next step is to transform these models automatically to TA models in the UPPAAL format. As both SDF^3 and UPPAAL utilises XML format, we have implemented a text-to-text transformation in the STARS tool-chain using *Epsilon Generation Language* (*EGL*) [RPKP08]. The generated TA models are already explained in Section 4.5.

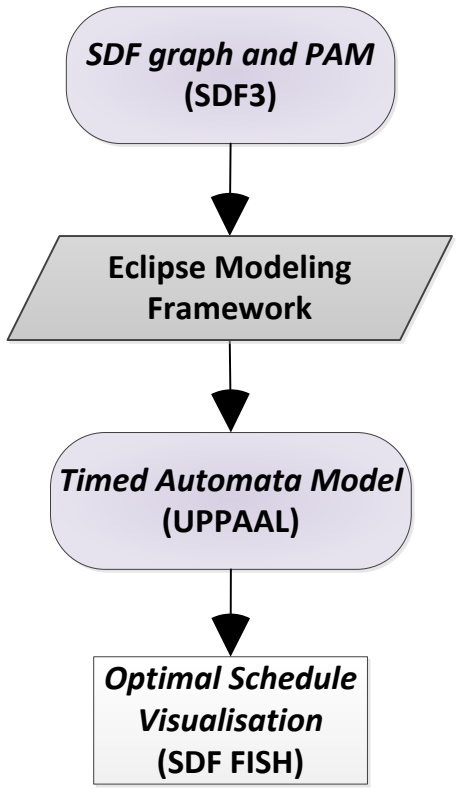


Figure 4.17: The workflow of the STARS tool-chain. After the SDF^3 model and PAM are transformed to UPPAAL using the Eclipse framework, we generate the optimal schedules using UPPAAL. Afterwards, we utilise SDF FISH to visualise the schedules.

4.8.3 Output

The optimisation problem of finding throughput-optimal schedules is encoded as a reachability query over TA. Afterwards, the model checker UPPAAL extracts a trace that satisfies the query. To visualise the traces generated by UPPAAL, we have developed a visualisation tool termed SDF FISH, having the following key features.

- Gantt chart style visualisation of SDF schedule.
- Zooming, filtering, and scaling features.
- Playing function to see changes occurring in the schedule over time.

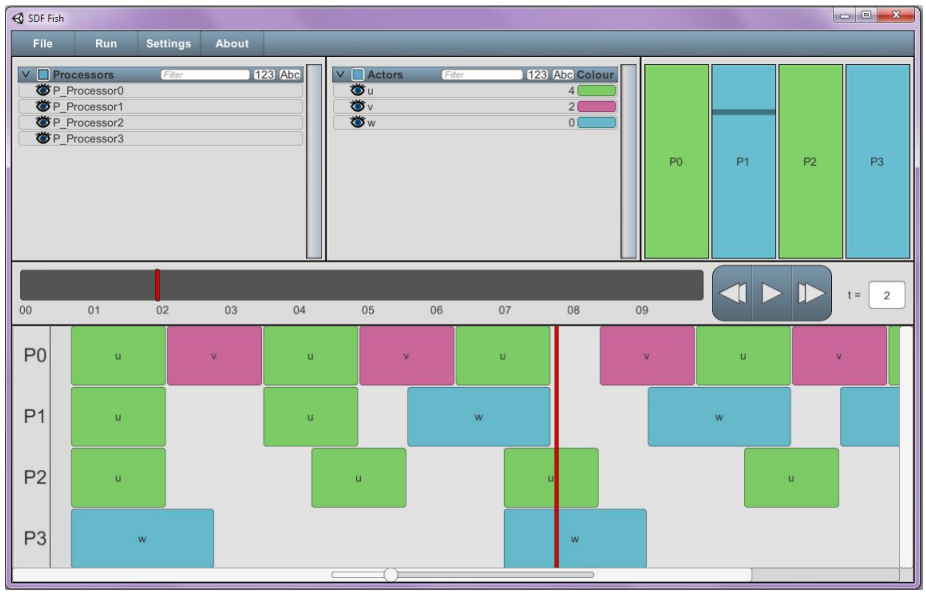


Figure 4.18: Screen shot of SDF FISH showing the schedule of the example in Section 4.3.3 on page 73. The lower part shows which actor is executed on which processor at what time. The playing function displays the progress of the complete schedule over time. On the upper part, we can see processors and actors involved in the schedule. Furthermore, we can also filter the schedule for the specific processors and actors. The upper part also shows the number of actor firings and processor utilisation.

- Various controls to navigate through the schedule such as number of times an actor has fired, and processor utilisation, until any given time.

The schedule visualisation of the example in Section 4.3.3 on page 73 in SDF FISH is given in Figure 4.18. As we can see in the lower part, the schedule shows which actor is executed on which processor at what time. Next to the time bar, we have a playing function to navigate through the complete schedule over time. On the upper left and middle part, we have all processors and actors involved in the schedule respectively. Using the filtering functions, the schedule can be visualised only for the specific processors and actors. The upper part also shows the number of actor firings and processor utilisation.

4.9 Conclusions

Despite the remarkable progress in the analysis of SDF graphs, compact methods for efficient scheduling of SDF graphs are still needed with an optimum trade-off between the maximum throughput and number of processors. In this chapter, we have demonstrated a novel throughput analysis technique for SDF-modelled

streaming applications on the given number of processors using TA. Moreover, our approach also generates optimal schedules. With experiments, we have compared our approach with SDF^3 which considers self-timed execution only to derive the throughput. We, on other hand, can handle any given number of processors. We also have introduced the STARS tool-chain which can automatically generate and visualise the schedules of the SDF graph modelled in SDF^3.

CHAPTER 5

Green Computing: Energy-Optimal Scheduling

Abstract

The previous chapter focused on resource-constrained scheduling of SDF graphs in such a way that the throughput is maximised. However, throughput is no longer the only performance metric for modern-day computer systems. In fact, a trend is emerging to trade raw performance for energy savings. Techniques like Dynamic Power Management (DPM, switching to low power state) and Dynamic Voltage and Frequency Scaling (DVFS, throttling processor frequency) help modern systems to reduce their power consumption while adhering to performance requirements. To balance flexibility and design complexity, the concept of Voltage and Frequency Islands (VFIs) was introduced for energy optimisation. It achieves fine-grained system-level energy management, by operating all processors in the same VFI at a common frequency/voltage.

This chapter presents a novel approach to compute an energy management strategy combining DPM and DVFS. In our approach, applications (modelled in synchronous dataflow, SDF) are mapped on heterogeneous multiprocessors (partitioned in voltage and frequency islands). We compute an energy-optimal schedule, meeting minimal throughput requirements. We demonstrate that the combination of DPM and DVFS provides an energy reduction beyond considering DVFS or DPM separately. Moreover, we show that by clustering processors in VFIs, DPM can be combined with any granularity of DVFS. Our approach uses model checking, by encoding the optimisation problem as a query over priced timed automata. The model checker UPPAAL CORA extracts a cost minimal trace, representing an energy minimal schedule. We illustrate our approach with several case studies introduced in Chapter 2 on commercially available hardware.

About this chapter: The current chapter is based on the paper "Green Computing: Power Optimisation of VFI-based Real-time Multiprocessor Dataflow Applications", which was published at DSD 2015 [AHSvdP15a]. An extended report on the work was published at University of Twente Eprints [AHSvdP15b]. The original paper largely remains the same.

5.1 Introduction

The power consumption of computing systems has increased exponentially [IP05]. Therefore, minimising power consumption has become one of the most critical, challenging and essential criteria for these systems. Over the past years, system-level power management based on the properties such as execution time of the tasks, frequency etc. has gained significant value and success [BBDM00, IP05, ZSB+13].

Power Reduction Techniques. The total power consumption of a processor is the sum of static (leakage) and dynamic power (in terms of switching activity). Two well-known techniques for power reduction in modern processors are Dynamic Power Management (DPM) [BBDM00] and Dynamic Voltage and Frequency Scaling (DVFS) [WWDS94]. DPM reduces the static power consumption, whereas DVFS is used to lower the dynamic power consumption.

Dynamic Power Management. DPM works on the principle of switching a processor to a low power state when it is not used, thus resulting in reduced power utilisation. For example, let us consider a processor of a typical mobile phone, having three power states, i.e., ON, INACTIVE, and OFF. If the processor runs in ON state, then the LCD and backlight of the phone is turned on. If the phone remains idle for some time, then the processor enters the INACTIVE state in which the backlight turns off but the LCD stays turned on. If the phone stays idle for some more time, then the LCD is turned off too (the OFF state). It is very commonly assumed by power optimisation methods in the literature that the transition overhead of switching to another power state is negligible [NPK+05]. However, this may not be the case in real-life applications, where there is always a non-negligible overhead [PPS+13]. We consider transition overheads while moving to a different power state. DPM is widely used; many processor manufacturers, such as Intel and AMD, have implemented an open standard for power management named *Advanced Configuration and Power Interface* [GHK14].

Dynamic Voltage and Frequency Scaling. On the other hand, DVFS [WWDS94] lowers the voltage and clock frequency at the expense of the execution time of a task. Power consumption of a processor scales linearly in frequency and quadratically in voltage. But, frequency and voltage also have a linear relation, therefore, when the clock frequency decreases, the voltage can also reduce, so that the power is reduced cubically. DVFS comes in two flavours, viz. local and global [MSH+11]. Local DVFS works on the principle that each processor has its own individual clock frequency/voltage, whereas all processors operate on the same clock frequency/voltage in the case of global DVFS. Local DVFS gives more freedom in choosing clock frequencies and is therefore more energy-efficient. However, local DVFS is expensive and complex to implement because it requires more than one clock domain. In contrast, global DVFS requires a simpler hardware design, but may lead to less reduction in power

consumption [MSH+11]. Several modern processors such as Intel Core i7 and NVIDIA Tegra 2 employ global DVFS [GHK14].

Voltage and Frequency Islands. To balance the energy efficiency and design complexity, the concept of *voltage and frequency islands* (VFIs) [HM07] was put forward. A VFI consist of a group of processors clustered together, and each VFI runs on a common clock frequency/voltage [PCL15]. The clock frequencies/-voltage supplies of a VFI may differ from other VFIs. Furthermore, different VFI partitions represent DVFS *policies* of different granularity. Recently, some modern multi-core processors, such as IBM Power 7 series, have adopted the option of VFIs [HWZ+12].

Shortcomings in Literature. While DPM and DVFS are popular power minimisation techniques, most of the earlier work [HMGM13, NMM+11, SDK13, ZSJ08] focuses on DVFS only, neglecting static power completely. On the contrary, modern processors have significant static power, which must be addressed. Hence, optimal energy minimisation cannot be guaranteed without considering both DVFS and DPM. Our work is the first to compute energy schedules for combined DVFS and DPM. Furthermore, with the help of VFIs, we combine DPM with a DVFS policy with *any* granularity, generalising local and global DVFS. This achieves fine-grained system-level energy management.

The second shortcoming in existing literature is addressing the applications where inter-task dependencies are modelled by directed acyclic graphs (DAGs), without analysing periodicity [dLJ06, LSWC08]; or frame-based periodic applications with no data dependencies between periods [DA12, GK13]. In real-time streaming applications, there are three challenges in implementing power management [HMGM13]. First, the schedules of these applications are typically infinite, making the problem scope infinite. Second, the iterations overlap in time and we have to deal with data dependencies within and across iterations. Last, performance constraints such as throughput are critical, and must be met. Hence, we cannot capture all semantics of real-time streaming applications using DAGs or frame-based models.

Our Approach. Alternatively, we use Synchronous Dataflow (SDF) [LM87b] as a model of computation (MoC) in this thesis. In addition to not considering the resource limitations, contemporary SDF analysis tools, e.g., SDF^3 [SGB06] also lack support for cost optimisation. Therefore, we propose an alternative, novel approach based on UPPAAL CORA [BLR05], the tool for Cost Optimal Reachability Analysis, using priced timed automata (PTA) as a modelling language. PTA extend timed automata [AD94] (for the modelling of time-critical systems and time constraints) with costs, which we use to model energy consumption. Furthermore, energy reduction techniques based on mathematical optimisation [HMGM13, NMM+11, WLL+11, GKA14] do not support quantitative model checking and evaluating user-defined properties. PTA also bridge this gap to achieve benefits over the range of analysable properties such as the absence of deadlocks and unboundedness, safety, liveness and reachability. Finally, PTA

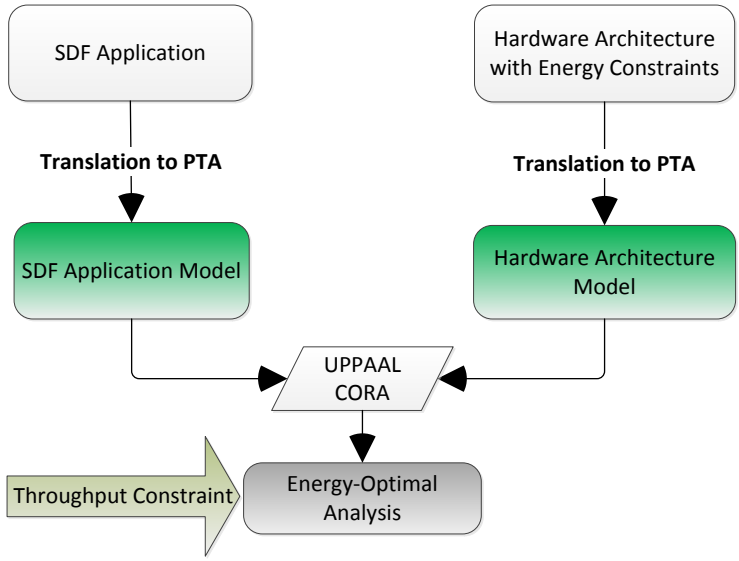


Figure 5.1: Our approach using priced timed automata. After both SDF application and hardware architecture with energy constraints are modelled as priced timed automata, UPPAAL CORA is used to derive energy-optimal schedules.

provide straightforward compositional and extendible modelling capabilities to system engineers, as opposed to mathematical optimisation approaches.

Methodology. Our approach as shown in Figure 5.1, takes three inputs: an SDF graph that models the application tasks; a platform model that describes the specifics of the hardware such as VFI partitions, frequency levels and power usage per processor; and a throughput constraint. We translate the SDF graph and the platform model to PTA using the model checker UPPAAL CORA. After defining the mapping of actors to processors, we utilise UPPAAL CORA to search the state-space. In this way, we compute an energy-optimal schedule that meets the constraint, utilising the dynamic and static slack in the application. The method can also be used to determine optimal VFI partitions in terms of design complexity and energy efficiency, facilitating system designers to build durable systems.

Contributions. The main contribution of this chapter is a fully automated technique to compute energy-optimal schedules. In particular, we demonstrate the following:

- We apply a combination of DPM and DVFS, confirming earlier results [DA12, GK13] that DPM and DVFS together result in lower energy consumption than considering only DVFS;

- Our method considers processors partitioned into VFIs; thus allowing to combine DPM, and DVFS policy with any granularity.
- We consider power overheads of transitions between different frequencies.
- Our approach is able to handle heterogeneous platforms, in which only specific processors can run a particular task.

Moreover, our technique is based on the solid semantic framework of priced timed automata, enabling the verification of functional system correctness.

We only consider discrete frequency and voltage levels in this work as real-life platforms can support only a limited set of discrete frequency and voltage levels [HMGM13].

Chapter Outline. Section 5.2 reviews related work. The power model considered in this chapter is introduced in Section 5.3. Section 5.4 explains SDF graphs and platform models equipped with energy constraints, with a help of an example. Different power reduction techniques are compared in Section 5.5, and Section 5.6 covers the translation of SDF graphs and platform models to PTA. The methodology of power optimisation of SDF graphs using UPPAAL CORA is explained in Section 5.7. Section 5.8 experimentally compares the results of different power optimisation techniques explained earlier, and Section 5.9 validates our approach via case studies. The tool-chain developed to support our work is described in Section 5.10. Finally, Section 5.11 draws conclusions.

5.2 Related Work

Considerable work has been done on power management. An extensive survey paper [IP05] outlines the research work in the field of algorithmic power management, but without reviewing any work done on SDF graphs. Another survey paper [ZSB+13] discusses several energy-cognizant scheduling techniques. All of the presented techniques do not evaluate effectiveness of optimal combination of global DVFS with scheduling. The novel methods for VFI-aware power optimisation are discussed in [JDP08] and [OMCM07]. It is assumed in these papers that task scheduling is finished beforehand, and therefore, task precedence is not considered. Whereas in practice, there are always precedence constraints due to inter-task data dependencies.

A state-of-the-art methods of applying DVFS only on SDF graphs is addressed in [NMM+11, SDK13, ZSJ08]. These papers, in comparison to ours, consider dynamic power only, and ignore static power which is non-negligible in modern processors. Moreover, work in [NMM+11] also requires to transform an SDF graphs to equivalent Homogeneous SDF (HSDF) graphs and model them with additional static ordering channels, which is not needed in our approach. Similarly, work in [ZSJ08] uses self-timed execution and static order firing, which means we need as many processors as actors, unlike real-life applications where there is always a constraint on the available number of processors. Therefore, this work is not scalable on any other hardware platform, where there are fewer processors than actors. Another method of throughput-constrained DVFS of SDF

graphs on a heterogeneous multiprocessor platform is proposed in [HMGM13]. The difference with our approach is that the work in [HMGM13] ignores transition overheads. Therefore, optimality cannot be guaranteed. Moreover, this paper also suffers from the limitation of static ordering.

More advanced approaches that combine DPM and DVFS are presented in [WLL+11, GKA14]. Unlike our method, these approaches discuss a specific energy optimisation policy where DPM and DVFS can be applied to each processor independently only. In contrast, we consider VFI-based hardware platforms where DPM and DVFS can be applied to any DVFS policy ranging from each processor having independent voltage/frequency level to all processors running at the same voltage/frequency level. Furthermore, these papers are restricted to acyclic applications only, which makes the problem scope simpler. The work in [DA12, GK13] describes an another novel algorithm for the optimal combination of DPM and DVFS. In comparison to our work, this technique is confined to global DVFS only, as it does not consider VFIs.

The next section introduces the power model used in this chapter.

5.3 Power Model

The clock frequency of a processor represents its speed. We assume that the speed of the processors scale linearly with the clock frequency. However, in practice, the relation between speed and clock frequency is not perfectly linear. The reason is that the computer memory is a separate device and it is often running on a different clock frequency. Therefore, the speed of the computer memory typically does not scale with the clock frequency of the processor [DA12].

Nevertheless, if measured at the maximum frequency, the round-trip time for a memory access in terms of processor clock cycles is at its highest. Memory access for the same task running at a lower frequency level is cheaper in terms of processor clock cycles. Therefore, our assumption made in this chapter that speed of the processors is linearly related to the clock frequencies, does not violate the deadline constraints of an application [GHK14].

The total power consumption by a processor is given by [dLJ06]:

$$P_{Tot} = P_D + P_S + P_{tr} \tag{5.1}$$

where P_D and P_S is the dynamic and static power usage of a processor respectively. The dynamic power is consumed due to the activity of the processor and is given by:

$$P_D = aCv_{dd}^2 f \tag{5.2}$$

where a is the circuit switching activity, C is the switched capacitance, v_{dd} is the supply voltage, and f is the operating frequency. Here, a and C are technology dependent. The static power is consumed independently of the processor activity and clock frequency. The static power is given by:

$$P_S = V_{dd} I_{subn} + |V_{bs}| I_j \tag{5.3}$$

where V_{dd} is the supply voltage, and rest of the parameters are fixed technology

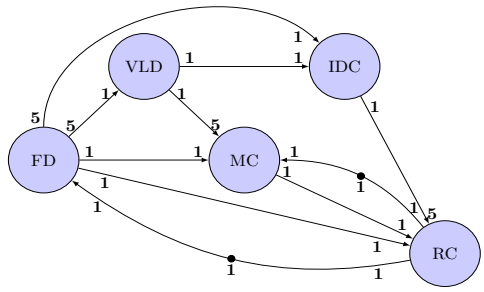


Figure 5.2: SDF graph of an MPEG-4 decoder

dependent. The transition overhead of transition from a certain frequency level to another is denoted by P_{tr}.

5.4 SDF Graphs with Energy Constraints

This section first recalls the definition of SDF graphs from Chapter 2. Furthermore, we also extend the *platform application model* defined earlier in Chapter 4 with energy constraints. The execution time of actors is dependent on the running frequency of processors in a platform application model (PAM). Therefore, we decided to include the execution time of actors in the definition of PAMs instead of SDF graphs, opposed to the definition of SDF graphs and PAMs in Chapter 4.

5.4.1 SDF Graphs

Definition 5.1. *An* SDF graph *is a tuple* $G = (A, D, \mathsf{Tok}_0)$ *where:*

- A *is a finite set of actors,*
- D *is a finite set of dependency channels* $D \subseteq A^2 \times \mathbb{N}^2$*, and*
- $\mathsf{Tok}_0 : D \to \mathbb{N}$ *denotes distribution of initial tokens in each channel, and*

The sets of input channels $In(a)$ *and* output channels $Out(a)$ *of an actor* $a \in A$ *are defined as:* $In(a) = \{(a', a, p, q) \in D | a' \in A \wedge p, q \in \mathbb{N}\}$ *and* $Out(a) = \{(a, b, p, q) \in D | b \in A \wedge p, q \in \mathbb{N}\}$. *The* consumption rate $CR(e)$ *and* production rate $PR(e)$ *of a channel* $e = (a, b, p, q) \in D$ *are defined as:* $CR(e) = q$ *and* $PR(e) = p$.

Informally, for all actors $a \in A$, if the number of tokens on every input channel $(a'_i, a, p_i, q_i) \in In(a)$ is greater than or equal to q_i, actor a fires and removes q_i tokens from every $In(a)$. The firing ends by producing p_i tokens on all $(a, b_i, p_i, q_i) \in Out(a)$.

Example 5.2. Figure 5.2 recalls the SDF graph of an MPEG-4 decoder from Chapter 2. The actors A={FD, VLD, IDC,RC, MC} represent individual tasks

performed in MPEG-4 decoding. For example, the frame detector (FD) models the part of the application that determines the frame type and the number of macro blocks to decode. The rest of the steps in MPEG-4 decoding are Variable Length Decoding (VLD), Inverse Discrete Cosine (IDC) Transformation, Motion Compensation (MC), and Reconstruction (RC) of the final video picture.

5.4.2 Platform Application Models

The Platform Application Model (PAM) models the multiprocessor platform where the application, modelled as SDF graph, is mapped on. We extend the definition of PAMs from Chapter 4 with a notion of power consumption. In particular, we extend with the following features.

- A partitioning of processors in voltage and frequency islands.
- Different frequency levels each processor can run on.
- Power consumed by a processor in a certain frequency, both when in use and when idle.
- Transition overheads required to switch between frequency levels.

Definition 5.3. *A* platform application model *(PAM) is a tuple* $\mathcal{P}=(\Pi, \zeta, F, P_{idle}, P_{occ}, P_{tr}, \tau_{act})$ *consisting of,*

- *a finite set of processors* Π. *We assume that* $\Pi = \{\pi_1, \ldots, \pi_n\}$ *is partitioned into disjoint blocks of voltage/frequency islands (VFIs) such that* $\bigcup \Pi_i = \Pi$, *and* $\Pi_i \cap \Pi_j = \emptyset$ *if* $i \neq j$,
- *a function* $\zeta : \Pi \to 2^A$ *indicating which processors can handle which actors,*
- *a finite set of discrete frequency levels available to all processors denoted by* $F = \{f_1, \ldots, f_m\}$ *such that* $f_1 < f_2 < \ldots < f_m$,
- *a function* $P_{occ} : \Pi \times F \to \mathbb{N}$ *denoting the power consumption (dynamic) of a processor operating at a certain frequency level* $f \in F$ *in the operating state,*
- *a function* $P_{idle} : \Pi \times F \to \mathbb{N}$ *assigning the power consumption (static) of a processor operating at a certain frequency level* $f \in F$ *in the idle state,*
- *a partial function* $P_{tr} : \Pi \times F^2 \nrightarrow \mathbb{N}$ *expressing the transition overhead from one frequency level* $f \in F$ *to next frequency level* $f \in F$ *for each processor* $\pi \in \Pi$, *and*
- *the valuation* $\tau_{act} : A \times F \to \mathbb{N}_{\geq 1}$ *defining the execution time of each actor* $a \in A$ *mapped on a processor at a certain frequency level* $f \in F$.

The notations f_i and Π_j represent i^{th} frequency level and j^{th} VFI respectively. We also use the notation $[\pi]$ to denote the VFI of a processor $\pi \in \Pi$.

Level	Voltage	Frequency	Level	Voltage	Frequency
1	1.2	1400	4	1.05	1128.7
2	1.15	1312.2	5	1.00	1032.7
3	1.10	1221.8			

Table 5.3: DVFS levels and corresponding CPU voltage and clock frequencies of Samsung Exynos 4210 processors

Example 5.4. Exynos 4210 [SAMa] is a state-of-the-art processor used in high-end mobile platforms such as Samsung Galaxy Note, Galaxy SII etc. Table 5.3 shows different DVFS levels, and corresponding CPU voltage (V) and clock frequency (MHz), of Samsung Exynos 4210 processors [PPS+13]. □

5.4.3 Example of SDF Graphs with Energy Constraints

In this subsection, we explain the aforementioned semantics of an SDF graph mapped on a processor application model by means of an example. Let us consider that the SDF graph of an MPEG-4 decoder shown in Figure 5.2 is mapped on four Samsung Exynos 4210 processors. The processors $\Pi = \{\pi_1, \pi_2, \pi_3, \pi_4\}$ are partitioned in three VFIs such that $\Pi_1 = \{\pi_1\}$, $\Pi_2 = \{\pi_2, \pi_3\}$ and $\Pi_3 = \{\pi_4\}$. Two DVFS levels (MHz) $\{f_1, f_2\} \in F$ taken from Table 5.3, i.e., $f_2 = 1400$ and $f_1 = 1032.7$ are available to all processors. The transition overhead (W) of all Exynos 4210 processors is, $P_{tr}(\pi, f_2, f_1) = 0.2$ and $P_{tr}(\pi, f_1, f_2) = 0.1$ [PPS+13]. Let us assume that all processors start at the highest frequency level, i.e., $f_2 \in F$. Table 5.4 shows the formation of VFIs and experimental power consumption against each frequency level. We also assume that the execution times (ms) of all actors $a \in A$ at the frequency level f_1 are rounded to the next integer. As $f_1 = 0.738 \times f_2$, $\tau_{act}(a, f_1) = \lceil \frac{\tau_{act}(a,f_2)}{0.738} \rceil$.

Figure 5.5 shows a schedule of our running example for a constraint of 125 frames per second (fps). To achieve 125 fps, MPEG-4 decoder completes the iteration in $\frac{1}{125}$= 8 ms. In this figure, grey and white coloured boxes denote, if a processor is running at the frequency level f_2 or f_1 respectively.

As we can see in Figure 5.5, the processor $\pi_1 \in \Pi$ changes its frequency level from $f_2 \in F$ to $f_1 \in F$ at t = 0 ms, thus incurring the transition overhead $P_{tr}(\pi_1, f_2, f_1) = 0.2$ W. From thereon, it operates at the frequency level $f_1 \in F$ for 5 ms. During this time interval, the actors FD $\in A$ and VLD $\in A$ are fired once on $\pi_1 \in \Pi$. At t = 5 ms, the processor $\pi_1 \in \Pi$ switches the frequency level from $f_1 \in F$ back to $f_2 \in F$ after incurring $P_{tr}(\pi_1, f_1, f_2) = 0.1$ W, and stays in the frequency level $f_2 \in F$ for the rest of the iteration. During this time interval, the actors IDC $\in A$ and RC $\in A$ claim $\pi_1 \in \Pi$ twice and once respectively. Thus per iteration, the processor $\pi_1 \in \Pi$ consumes dynamic energy for 8 ms. As the processor $\pi_1 \in \Pi$ does not remain idle during the iteration, it does not consume any static energy. The total energy consumption (mWs) per iteration of the

Processor	VFI	Voltage(V)	Frequency(MHz)	P_{idle}(W)	P_{occ}(W)
π_1	Π_1	1.2	1400	0.1	4.6
		1.00	1032.7	0.4	1.8
π_2	Π_2	1.2	1400	0.1	4.6
		1.00	1032.7	0.4	1.8
π_3	Π_2	1.2	1400	0.1	4.6
		1.00	1032.7	0.4	1.8
π_4	Π_3	1.2	1400	0.1	4.6
		1.00	1032.7	0.4	1.8

Table 5.4: Description of the platform containing four Samsung Exynos 4210 processors π_1, π_2, π_3, and π_4. These processors are partitioned into three VFIs Π_1, Π_2, and Π_3.

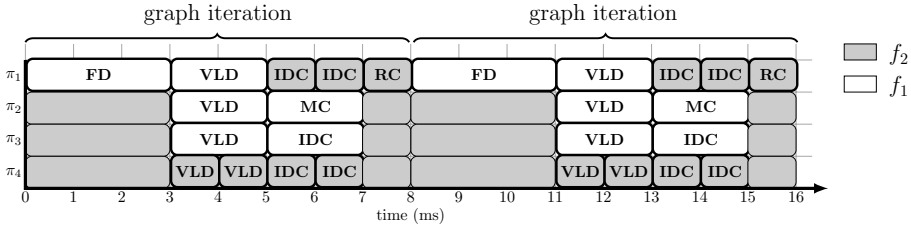


Figure 5.5: An example schedule of MPEG-4 decoder in Figure 5.2 on four processors π_1, π_2, π_3, and π_4. The grey and white coloured boxes denote, if a processor is running at the frequency f_2 or f_1 respectively

processor $\pi_1 \in \Pi$ is

$$E_{Tot} = E_S + E_D + E_{tr}$$

$$E_{Tot} = P_{idle} \times 0 + P_{occ}(\pi_1, f_2) \times 3 + P_{occ}(\pi_1, f_1) \times 5 + P_{tr}(\pi_1, f_2, f_1) + P_{tr}(\pi_1, f_1, f_2)$$

$$E_{Tot} = 0 + 4.6 \times 3 + 1.8 \times 5 + 0.2 + 0.1 = 23.1$$

In the same fashion, we can calculate energy consumption per iteration for each processor, which gives us the total energy consumption equal to 57.7 mWs per iteration.

5.4.4 Semantics of SDF Graphs with Energy Constraints

The dynamic behaviour of an SDF graph G mapped on a PAM $\mathcal{P}$ can naturally be understood in terms of a labelled transition system (LTS) $\mathcal{LTS}(G_{\mathcal{P}})$. Below, we define $\mathcal{LTS}(G_{\mathcal{P}})$ by giving its states and transitions.

Definition 5.5. *A* state *is a tuple* $(Tok, status, freq, TuC, TotPow)$ *with the*

following components.

- *channel quantity* $Tok : D \to \mathbb{N}$ *associates with each channel the number of tokens currently present in that channel, and*
- $status : \Pi \to \{idle, occup\}$ *and* $freq : \Pi \to F$ *associates with each processor* $\pi \in \Pi$*, whether it is idle or occupied, and its current frequency level* $f \in F$*.*
- *To observe the progress of time,* $TuC : \Pi \to \mathbb{N}$ *records for each processor the remaining execution time required to complete its current task.*
- $TotPow : \Pi \to \mathbb{N}$ *records the total accumulated power consumption for each processor.*

The initial state $(Tok_0, status_0, freq_0, TuC_0, TotPow_0)$ *is given by* $status_0(\pi) = idle$*,* $freq_0(\pi) = f_m$*,* $TuC(\pi) = 0$*,* $TotPow_0(\pi) = 0$ *for all* $\pi \in \Pi$*.*

Example 5.6. The initial state of the example explain in Section 5.4.3 is given by $(Tok_0, status_0, freq_0, TuC_0, TotPow_0) = ((0,0,0,0,0,0,0,0,1,1), (idle, idle, idle, idle), (f_2, f_2, f_2, f_2), (0,0,0,0), (0,0,0,0))$. Here, the initial tokens in all channels are represented by Tok_0. The initial availability of processors and their active frequency level is given by $status_0$ and $freq_0$ respectively. Similarly, TuC_0 and $TotPow_0$ represents the initial, remaining execution times and power consumption, of processors respectively. □

The transitions in $\mathcal{LTS}(G_{\mathcal{P}})$ are given in the following.

Definition 5.7. *A* transition *from state* $(Tok_1, status_1, freq_1, TuC_1, TotPow_1)$ *to* $(Tok_2, status_2, freq_2, TuC_2, TotPow_2)$ *is denoted as,*

$$(Tok_1, status_1, freq_1, TuC_1, TotPow_1) \xrightarrow{\kappa} (Tok_2, status_2, freq_2, TuC_2, TotPow_2)$$

The label κ *is defined as* $\kappa \in (A \times \Pi \times F \times \{start, end\}) \cup \{tick\} \cup (\Pi \times F \times F \times jump)$ *and corresponds to the type of transition.*

- *Label* $\kappa = (a, \pi, f, start)$ *denotes mapping and starting of an firing of an actor* $a \in A$ *on a processor* $\pi \in \Pi$ *at a frequency level* $f \in F$*. This transition may occur if*
  - $Tok_1(d) \geq CR(d)$ *for all* $d \in In(a)$*. That is, all input channels* $d \in D$ *have sufficiently many tokens.*
  - $status_1(\pi) = idle$*, i.e., the processor* π *is currently unoccupied.*
  - $freq_1(\pi') = f$ *for all* $\pi' \in [\pi]$*, i.e., the active frequency level of all processors in VFI* $[\pi]$ *is* $f \in F$*, and*
  - $a \in \zeta(\pi)$*, i.e., if the actor* a *can be mapped on the processor* π *or not.*

  This transition results in a new state $(Tok_2, status_2, freq_2, TuC_2, TotPow_2)$ *given by,*
  - $Tok_2(d) = Tok_1(d) - CR(d)$ *for all* $d \in In(a)$*. That is,* $CR(d)$ *tokens are removed from each incoming channel.*

- $status_2(\pi') = status_1(\pi')$ *and* $status_2(\pi) = occup$ *for all* $\pi' \neq \pi$. *That is, the processor* $\pi \in \Pi$ *is claimed.*
- $freq_2 = freq_1$, *i.e., the active frequency level of all processors does not change,*
- $TuC_2(\pi) = \tau_{act}(a, f)$, *i.e.,* $\tau_{act}(a, f)$ *is attached to the processor* π, *and*
- $TotPow_2 = TotPow_1$, *i.e., this transition does not cost any power.*

Example 5.8. The actor *FD* in the example given in Section 5.4.3 takes the transition $(FD, f_2, \pi_1, start)$ at time t = 0 ms. As a result, one token is subtracted from the channel $RC - FD$. □

- *Label* $\kappa = (a, \pi, f, end)$ *denotes the ending of a firing by an actor* $a \in A$ *and releasing a processor* $\pi \in \Pi$ *operating at a frequency level* $f \in F$. *This transition may occur if,*
  - $TuC_1(\pi) = 0$, *i.e., the actor* $a \in A$ *has finished its execution.*

  This transition results in a new state $(Tok_2, status_2, freq_2, TuC_2, TotPow_2)$ *given by,*
  - $Tok_2(d) = Tok_1(d) + PR(d)$ *for all* $d \in Out(a)$. *That is,* $PR(d)$ *tokens are produced on all output channels.*
  - $TuC_2 = TuC_1$,
  - $status_2(\pi') = status_1(\pi')$, *and* $status_2(\pi) = idle$ *for all* $\pi' \neq \pi$. *That is, the processor* $\pi \in \Pi$ *is released.*
  - $freq_2 = freq_1$, *i.e., the active frequency level of all processors does not change, and*
  - $TotPow_2 = TotPow_1$, *i.e., this transition does not cost any power.*

Example 5.9. In the example given in Section 5.4.3, the actor *FD* takes the transition (FD, f_2, π_1, end) at time t = 3 ms. As a result, five tokens are produced on the channels $FD - IDC$ and $FD - VLD$, and one token is produced on the channels $FD - MC$ and $FD - RC$. □

- *Label* $\kappa = tick$ *denotes a clock tick transition. This transition is enabled if,*
  - $TuC_1(\pi') \neq 0$ *for all* $\pi' \in \Pi$. *That is, no end transition is enabled.*

  This transition results in a new state $(Tok_2, status_2, freq_2, TuC_2, TotPow_2)$. *For all* $d' \in D$ *and* $\pi' \in \Pi$, *the new state is given by,*
  - $Tok_2(d') = Tok_1(d')$,
  - $status_2(\pi') = status_1(\pi')$,
  - $freq_2(\pi') = freq_1(\pi')$,
  - $TuC_2(\pi') = TuC_1(\pi') - 1$, *i.e., the remaining execution time assigned to the processors is decreased by 1,*

- *if* $status_1(\pi') = occup$, *then* $TotPow_2(\pi') = TotPow_1(\pi') + P_{occ}(\pi', freq_1(\pi'))$, *and*
- *if* $status_1(\pi') = idle$, *then* $TotPow_2(\pi') = TotPow_1(\pi') + P_{idle}(\pi', freq_1(\pi'))$.

Example 5.10. In our running example, there are three *tick* transitions between t = 0 ms and t = 3 ms on the processor $\pi_1 \in \Pi$, because no end transition is enabled in that period. At t = 0 ms, the execution time of the actor *FD*, i.e., $\tau_{act}(FD, f_2)$ is attached to the processor π_1. After three *tick* transitions, the remaining execution time assigned to the processor π_1 equals 0, and therefore an end transition is taken at t = 3 ms. □

- *Label* $\kappa = (\Pi_j, f_i, f', jump)$ *denotes a transition of all processors* $\pi' \in \Pi_j$ *running at a frequency level* $f_i \in F$, *to another frequency level* $f' \in F$ *such that* $f' = f_{i+1}$ *or* $f' = f_{i-1}$. *This transition is enabled if,*
  - *for all* $\pi' \in \Pi_j$, $freq_1(\pi') = f_i$ *and* $status_1(\pi') = idle$, *i.e., the processors in the same VFI can change to another frequency level only if they all are in the idle state at the same frequency level.*

  This transition results in a new state $(Tok_2, status_2, freq_2, TuC_2, TotPow_2)$ *given by,*
  - $Tok_2(d) = Tok_1(d)$ *for all* $d \in D$. *That is, the token distribution does not change.*
  - $status_2(\pi') = idle$ *and* $freq_2(\pi') = f'$ *for all* $\pi' \in \Pi_j$. *That is, the active frequency level of all processors in the same VFI changes to* $f' \in F$.
  - $TuC_2 = TuC_1$, *and*
  - $TotPow_2(\pi') = TotPow_1(\pi') + P_{tr}(\pi')$ *for all* $\pi' \in \Pi_j$. *That is, the transition overhead of all processors belonging to the same VFI is incurred.*

  Example 5.11. In Figure 5.5, there is one *jump* transition at time t = 0 ms, i.e., $(\Pi_1, f_2, f_1, jump)$. □

5.5 Comparison of Energy Optimisation Methods

This section illustrates the importance of considering DPM along with DVFS, with the help of a non-trivial observation. Furthermore, we explain how VFIs allow us to achieve fine-grained energy optimisation, by combining DPM with any granularity of DVFS. Let us consider a real-time periodic application mapped on a single processor. Figure 5.6 shows the behaviour of static (E_S) and dynamic (E_D) energy consumption of the processor as a function of processor frequency for the execution of an entire iteration. Note that E_S also includes transition overheads. The minimum frequency at which the task can meet its deadline is denoted by f_a. Similarly, f^* denotes the minimum frequency at which there

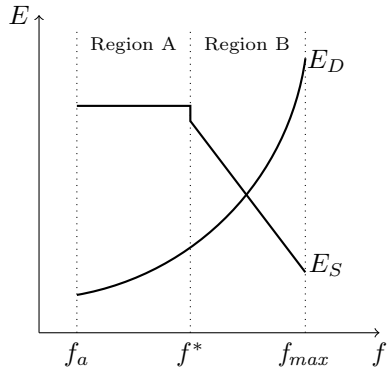


Figure 5.6: Static (E_S) and dynamic (E_D) energy as a function of frequency

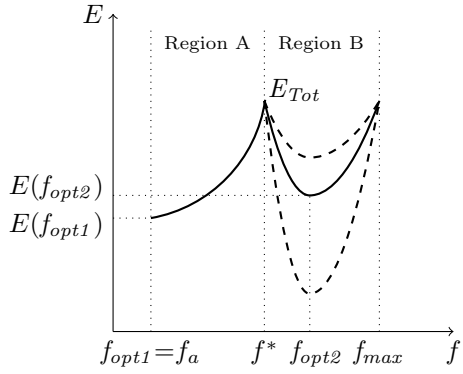


Figure 5.7: Total energy (E_{Tot}) as a function of frequency

is enough slack for the processor to move to the low power state. Thus, the processor can only move to the low power state, if its frequency is no less than f^*. Otherwise, it will not be able to meet the deadline.

As explained earlier, E_D increases cubically with the increase of frequency. However, E_S shows varying patterns. In Region A where $f_a \leq f < f^*$, the idle period of the processor is too short to allow it to move to the low power state where static energy consumption is lower. Therefore, E_S is higher and constant in Region A. However, as frequency reaches f^*, slack, i.e., the idle period of the processor increases, allowing the transition to less static power consuming states. Thus, E_S drops down at $f = f^*$. As frequency increases beyond f^* in Region B, the idle period of the processor increases further in linear fashion, leading to switching to deeper sleep states by the processor. Without loss of generality, if we assume that transition overhead of switching to deeper low power states also increases linearly, we get linear decrease of E_S with the increase of frequency in Region B, as shown in Figure 5.6.

Figure 5.7 shows $E_{Tot} = E_S + E_D$, as a function of processor frequency. In Region A where E_{Tot} grows with the increasing frequency, local minimum f_{opt1} of E_{Tot} is $f_{opt1} = f_a$. Whereas, in Region B, E_{Tot} decreases with the increasing frequency. The local minimum f_{opt2} of E_{Tot} in Region B is $f_{opt2} \in [f^*, f_{max}]$. Depending on power consumption of low power states, and transition overheads, steepness of E_S can increase or decrease in Region B. As a result, the minimum value of E_{Tot} can have different values in Region B, as shown by dashed lines.

As we have seen that local optimal frequencies to minimise E_{Tot} in both regions are well defined. However, *there is no a priori reason that global minimum of E_{Tot} should lie in Region A or Region B*. Depending on the power consumption values of the processor and deadline of the application, the global optimal frequency can be either in Region A or B.

Alternatively, if we do not consider DPM, E_S in Region B remains same as A, and consequently, E_{Tot} increases in Region B as well. Therefore, we can safely conclude that we must consider *both* DPM and DVFS to determine the

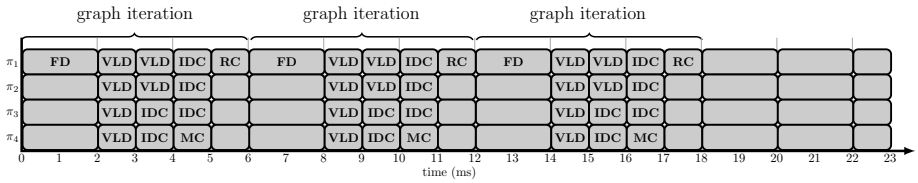


Figure 5.8: Energy-optimal execution wrt `No − PowerOpt`. The processors remain idle for last 5 ms resulting in dynamic slack.

Voltage(V)	**Frequency(MHz)**	**GDVFS**		**GDVFS+DPM**	
		P_{idle}(W)	P_{occ}(W)	P_{idle}(W)	P_{occ}(W)
1.2	1400	0.4	4.6	0.1	4.6
1.00	1032.7	0.4	1.8	0.4	1.8

Table 5.9: Frequency and voltage levels considered for `GDVFS` and `GDVFS + DPM`

optimal energy consumption. We can generalise this result for multiprocessors also. Moreover, partitioning processors into VFIs enable us to assign frequency per partition, rather than running all processors at the same frequency.

To illustrate earlier arguments, let us consider an example of an MPEG-4 decoder shown in Figure 5.2 on page 101, mapped on the platform containing four Samsung Exynos 4210 processors, i.e., $\Pi = \{\pi_1, \pi_2, \pi_3, \pi_4\}$. For the deadline of completing three graph iterations within 23 ms, we consider the following scenarios.

- **Case 0**: Without Power Optimisation (`No − PowerOpt`)

  Let us assume that the processors do not utilise any power management technique. The only frequency $f \in F$ available to the processors is $f =$ 1400 MHz. The idle (static) and operating (dynamic) power consumption at $f = 1400$ MHz is $P_{idle}(\pi, f)$=0.4 W and $P_{occ}(\pi, f)$=4.8 W respectively. Figure 5.8 shows the optimal execution of this case, where we can see that the constraint of finishing three graph iterations is met well before the deadline, and the processors remain idle for the rest of the time resulting in dynamic slack. Hence, DVFS is needed to minimise dynamic slack. The total energy consumption of this case is 204.2 mWs.

- **Case 1**: Global DVFS only (`GDVFS`)

  Now, to introduce DVFS in processors, we add an extra frequency level (MHz), i.e., $\{f_1, f_2\} \in F$ such that $f_2 = 1400$ and $f_1 = 1032.7$. In this case, the processors employ DVFS only, without considering DPM and VFIs. Table 5.9 shows the idle (static) and operating (dynamic) power consumption at both frequencies. Note that, idle power consumption of all processors $\pi \in \Pi$ is constant at both frequencies, i.e., $P_{idle}(\pi, f_2)$=$P_{idle}(\pi, f_1)$=0.4 W.

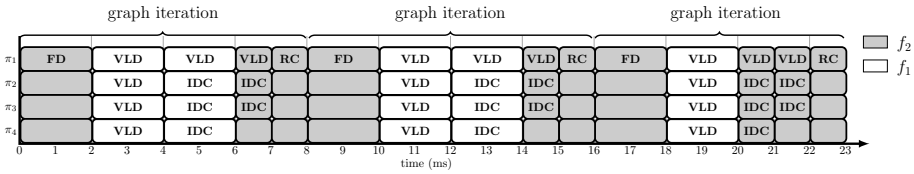


Figure 5.10: Energy-optimal execution wrt GDVFS. The processors consume high static power while being idle, leading to static slack.

Recall that GDVFS assumes one VFI, i.e., $\Pi_1 = \{\pi_1, \pi_2, \pi_3, \pi_4\}$. The optimal execution of this scenario is shown in Figure 5.10. As we can see in Figure 5.10, the constraint of finishing three graph iterations is fulfilled exactly at the deadline, as opposed to $\mathtt{No-PowerOpt}$. Thus, DVFS helps to reduce dynamic slack. As a result, the total energy consumption drops to 185.2 mWs from 204.2 mWs. However, in case of GDVFS, the processors consume high static power while being idle, leading to static slack. Therefore, we must utilise DPM to reduce static slack.

- **Case 2**: Global DVFS + DPM ($\mathtt{GDVFS+DPM}$)

  In order to allow processors benefit from both DPM and DVFS, we introduce a low power state, i.e., the idle power consumption of all processors $\pi \in \Pi$ at frequency level $f_2 = 1400$ MHz is changed to $P_{idle}(\pi, f_2)$=0.1 W because more idle time allows DPM. However, the operating power consumption of all processors $\pi \in \Pi$ at both frequencies, i.e., $P_{occ}(\pi, f_2)$ and $P_{occ}(\pi, f_1)$ remains same as GDVFS, as given in Table 5.9. The transition overhead (W) of all processors $\pi \in \Pi$ is, $P_{tr}(\pi, f_2, f_1) = 0.2$ and $P_{tr}(\pi, f_1, f_2) = 0.1$. In this case, the schedule remains the same. However, the total energy consumption drops significantly to 179.8 mWs. Hence, it shows that optimality of energy minimisation can only be guaranteed by considering both DPM and DVFS. However, as we may observe, all processors run at the same frequency in $\mathtt{GDVFS+DPM}$, which might be unnecessary. Instead, we may partition processors into VFIs so that only required processors run at the same frequency, and others may run at the different frequency.

- **Case 3**: DVFS + DPM with 2 VFIs ($\mathtt{DVFS+DPM-2}$)

  In this scenario, we partition processors into two VFIs such that $\Pi_1 = \{\pi_1, \pi_2\}$ and $\Pi_2 = \{\pi_3, \pi_4\}$, while utilising both DVFS and DPM. The power consumption values of processors at both frequencies remain the same as in Case 2. As a result, the total energy consumption reduces to 179.2 mWs, demonstrating the effectiveness of VFIs to achieve fine-grained energy management. The optimal execution of this case is shown in Figure 5.11.

- **Case 4**: DVFS + DPM with 3 VFIs ($\mathtt{DVFS+DPM-3}$)

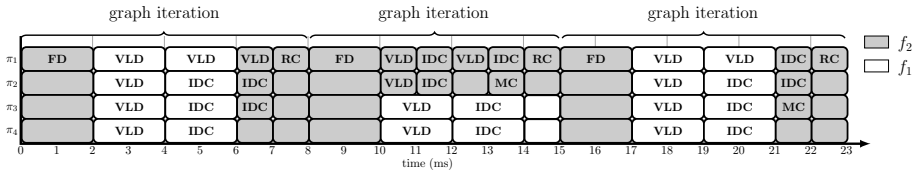


Figure 5.11: Energy-optimal execution wrt $\texttt{DVFS} + \texttt{DPM} - 2$. The processors are partitioned into two VFIs.

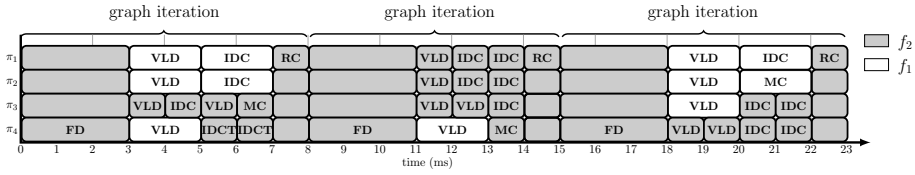


Figure 5.12: Energy-optimal execution wrt $\texttt{DVFS} + \texttt{DPM} - 3$. The processors are partitioned into three VFIs.

The total energy consumption drops further to 176.5 mWs, if we partition processors into three VFIs such that $\Pi_1 = \{\pi_1\}$, $\Pi_2 = \{\pi_2\}$ and $\Pi_3 = \{\pi_3, \pi_4\}$. Figure 5.12 shows the optimal execution of Case 4, i.e., $\texttt{DVFS} + \texttt{DPM} - 3$.

5.6 From SDF Graphs and Platform Application Models to Priced Timed Automata

Similar to previous chapter, our framework consists of separate models of an SDF graph and the processor application model. In this section, we describe the translation of an SDF graph along with a PAM to PTA using UPPAAL CORA.

Given an SDF graph $G = (A, D, \mathsf{Tok}_0, \tau)$ mapped on a PAM $\mathcal{P} = (\Pi, \zeta, F, P_{idle}, P_{occ}, P_{tr}, \tau_{act})$, we generate a parallel composition of PTA:

$$A_G \| Processor_1 \|, \ldots, \| Processor_n \| Scheduler.$$

PTA models of the example given in Section 5.4.3 is shown in Figure 5.13 on page 116. Here, the automaton A_G models actors and channels of an SDF graph, as shown in Figure 5.13a. The PTA $Processor_1, \ldots, Processor_n$ model processors $\Pi = \{\pi_1, \ldots, \pi_n\}$, as shown in Figure 5.13b. Figure 5.13c presents the automaton of *Scheduler*, that decides when to switch the frequency level of all processors in the same VFI. We assume that the underlying LTS of G is given by $(S, Lab, \rightarrow_G)$ where $S = (Tok, status, freq, TuC, TotPow)$ denotes the states, $Lab = \kappa$ denotes the labels, and $\rightarrow_G \subseteq S \times Lab \times S$ depicts the transitions.

Priced Timed Automaton A_G. The automaton A_G models the SDF graph G. Given G and $\mathcal{P}$, the automaton A_G is defined as,

$$A_G = (L, l^0, Act, E, P, Inv)$$

All components of A_G are explained as follows.

- The location $L = l_0 = \{\mathsf{Initial}\}$ is the only location in our SDF graph model.
- The action set $Act = \{\mathsf{fire!}, \mathsf{end?}\}$ contains two parametrised actions, i.e., fire! (exclamation mark signifies a sending operation) and end? (question mark signifies a receiving operation) to synchronise with the PTA $Processor_1, \ldots, Processor_n$. For each processor $\pi \in \Pi$ and $a \in A$, $\mathsf{fire}[\pi][\mathsf{a}]$ represents the start of the firing of an actor a on a processor π, and $\mathsf{end}[\pi][\mathsf{a}]$ represents its firing. The action $\mathsf{fire}[\pi][\mathsf{a}]$ is enabled if the incoming channels of $a \in A$ have sufficient tokens.
- For each $a \in A$ and $d \in D$, the edge set E contains two edges such that:
  - $\mathsf{Initial} \xrightarrow{Tok(d) \geq CR(d):\ \mathsf{fire}[\pi][\mathsf{a}]!,\ Tok(d) = Tok(d) - CR(d)} \mathsf{Initial}$
  - $\mathsf{Initial} \xrightarrow{true:\ \mathsf{end}[\pi][a]?,\ Tok(d) := Tok(d) + PR(d)} \mathsf{Initial}$

    Here, $Tok(d) \geq CR(d)$ refers to a *guard* and it signifies that tokens on all input edges $d \in In(a)$ of an actor $a \in A$ must be greater than or equal to their consumption rate in order to take the action fire!. As a result of taking the action fire!, tokens on all input edges $d \in In(a)$ of an actor $a \in A$ are subtracted, i.e., $Tok(d) = Tok(d) - CR(d)$. Similarly, by taking the action end?, actor firing is completed and tokens are produced on all output edges $d \in Out(a)$ of an actor $a \in A$, i.e., $Tok(d) = Tok(d) + PR(d)$.
- The automaton does not contain any costs.
- We do not have any clocks and invariants in A_G. Therefore, *Inv:* $L \rightarrow B(C)$ and $Inv(l^0) = true$.

The automaton A_G contains a number of variables: for each channel from actors $a \in A$ to $b \in B$, an integer variable $\mathsf{buff\_a2b} = Tok(\mathsf{a}, \mathsf{b}, \mathsf{p}, \mathsf{q})$ containing the number of tokens in the channel from a to b. The variable counter_a counts how many times actor $a \in A$ has been fired in an execution. Initially, counter_a $= 0$ and $\mathsf{buff\_a2b} = Tok_0(\mathsf{a}, \mathsf{b}, \mathsf{p}, \mathsf{q})$ contains the number of tokens in the initial distribution of the channel (a, b, p, q).

The action $\mathsf{fire}[\pi][\mathsf{a}]$ consumes, from each input channel $(b, a, p, q) \in In(a)$ in G, the q tokens from the variable buff_b2a, and is carried out by the function $\mathsf{consume}(\mathsf{buff\_b2a}, \mathsf{q})$. The action end[i][a] adds, for each actor $a \in A$ and output channel $(a, b, p, q) \in Out(a)$ in G, the p tokens on the variable buff_a2b by carrying out the function $\mathsf{produce}(\mathsf{buff\_a2b}, \mathsf{p})$.

The function estimate() provides an estimate of lower bound on the remaining cost, which is used to improve the performance of UPPAAL CORA.

Priced Timed Automata Processor$_j$. The priced timed automata (PTA) *Processor$_j$* model PAMs. For each $\pi_j \in \Pi$, we define *Processor$_j$* PTA as follows.

$$Processor_j = (L_j, l_j^0, Act_j, E_j, P_j, Inv_j)$$

All components of *Processor$_j$* are explained as follows.

- For each frequency level $f_i \in F$, we include both an idle state and an active state running on that frequency level in the PTA *Processor$_j$*. Thus, for each $a \in \zeta(\pi_j)$ and $F = \{f_1, \ldots, f_m\}$ such that $f_1 < f_2 < \ldots < f_m$, let $L_j = \{\mathsf{Idle\_f1}, \ldots, \mathsf{Idle\_fm}, \mathsf{InUse\_a\_f1}, \ldots, \mathsf{InUse\_a\_fm}\}$ indicating that the processor $\pi_j \in \Pi$ is currently used by the actor $a \in A$ in the frequency level $f_i \in F$, either in idle or running state.

- For $F = \{f_1, \ldots, f_m\}$ such that $f_1 < f_2 < \ldots < f_m$, the initial location is given by $l_j^0 = \mathsf{Idle\_fm}$. This explains that a processor $\pi \in \Pi$ always starts at the highest frequency level $f_m \in F$.

- The action set $Act_j = \{\mathsf{fire?}, \mathsf{end!}, \mathsf{jump\_ik?}\}$ contains three actions fire?, end! and jump_ik?. The actions fire? and end! in Act_j are parametrised with processor and actor ids, and synchronise with A_G. The action jump_ik? in Act_j is parametrised with the VFI id. For all $f_i, f_k \in F$, and $\pi_j \in \Pi_y$, the broadcast action $\mathsf{jump\_ik}[y]$ synchronises the automata *Processor*$_1, \ldots,$ *Processor*$_n$ with the automaton *Scheduler*, to switch all processors in the VFI $[\pi_j]$ from the frequency level f_i to f_k.

- For each $\pi \in \Pi$, $a \in \zeta(\pi)$ and $f_i \in F$, the edge set E_j contains two edges such that:

  – $\mathsf{Idle\_fi} \xrightarrow{true:\, \mathsf{fire}[\pi][\mathsf{a}]?,\, \mathsf{x}=0} \mathsf{InUse\_a\_fi}$, and

  – $\mathsf{InUse\_a\_fi} \xrightarrow{x=\tau_{act}(a,f_i):\, \mathsf{end}[\pi][\mathsf{a}]!,\, \varnothing} \mathsf{Idle\_fi}$.

  The action $\mathsf{fire}[\pi][\mathsf{a}]$ is enabled in the idle state Idle_fi and leads to the location InUse_a_fi. Thus, $\mathsf{fire}[\pi][\mathsf{a}]$ claims the processor $\pi \in \Pi$ at the frequency level $f_i \in F$, so that any other firing cannot occur on $\pi \in \Pi$ before the current firing of $a \in A$ is finished. As each location InUse_a_fi has an invariant $Inv_j(\mathsf{InUse\_a\_fi}) \leq \tau_{act}(\mathsf{a}, \mathsf{f_i})$, the automaton can stay in InUse_a_fi for at most the execution time of actor $a \in A$ at frequency level $f_i \in F$, i.e., $\tau_{act}(a, f_i)$. If $x = \tau_{act}(a, f_i)$, the system has to leave InUse_a_fi at exactly the execution time of actor $a \in A$ at frequency level $f_i \in F$, by taking the $\mathsf{end}[\pi][\mathsf{a}]$ action. In this way, A_G is notified that the execution of $a \in A$ has ended, so that A_G updates the variables.

- For $F = \{f_1, \ldots, f_k, f_i\}$ such that $f_1 < f_2 < \ldots < f_k < f_i$, and $\pi_j \in \Pi_y$, the edge set E_j has the following edges $E_{broad} \in E$ for handling broadcast such that:

  – $\mathsf{Idle\_fi} \xrightarrow{true:\, \mathsf{jump\_ik}[y]?,\, \varnothing} \mathsf{Idle\_fk}$,

  - $\mathsf{Idle\_fk} \xrightarrow{true:\, \mathsf{jump\_ki}[y]?,\, \varnothing} \mathsf{Idle\_fi}$,
    $\vdots$
  - $\mathsf{Idle\_f1} \xrightarrow{true:\, \mathsf{jump\_12}[y]?,\, \varnothing} \mathsf{Idle\_f2}$

- For each actor $a \in \zeta(\pi)$ and frequency level $f_i \in F$, the costs to stay in idle or running states are given as $P_j(\mathsf{Idle\_fi}) = P_{idle}(\pi, \mathsf{f_i})$ and $P_j(\mathsf{InUse\_a\_fi}) = P_{occ}(\pi, \mathsf{f_i})$. Furthermore, for all $f_i, f_k \in F$, $\pi_j \in \Pi_y$, and $E_{broad} \in E$, the cost of transition from one frequency level to another is given as $P_j(\mathsf{Idle\_fi} \xrightarrow{true:\, \mathsf{jump\_ik}[y]?,\varnothing} \mathsf{Idle\_fk}) = P_{tr}(\pi_\mathsf{j}, \mathsf{f_i}, \mathsf{f_k})$.
- For each actor $a \in \zeta(\pi)$ and frequency level $f_i \in F$, the invariants are given as $Inv_j(\mathsf{Idle\_fi}) = true$, and $Inv_j(\mathsf{InUse\_a\_fi}) \leq \tau_{act}(a, f_i)$ enforcing the system to stay in $\mathsf{InUse\_a\_fi}$ for at most the execution time $\tau_{act}(a, f_i)$.

Please note that $Processor_j$ contains exactly one clock x_j; since clocks in UPPAAL CORA are local, we can abbreviate x_j by x. A separate clock variable global observes the overall time progress. Moreover, as we only can have integer costs, all values of power consumption are multiplied by 10 in the UPPAAL CORA model in Figure 5.13 on page 116.

For all $\pi_j \in \Pi_y$, $Processor_j$ has a variable freq_lev[y] to count processors in the running state. Initially, freq_lev[y] = 0 for all $\pi_j \in \Pi_y$. If a processor $\pi_j \in \Pi_y$ is claimed by an actor $a \in A$, the counter freq_lev[y] is incremented by one. Similarly, if a processor $\pi_j \in \Pi_j$ is released, the value of the counter freq_lev[y] is reduced by one. For all $\pi_j \in \Pi_y$, $Processor_j$ has another variable curr_freq[y] that determines the current frequency level of all $\pi_j \in \Pi_y$. Initially, for $F = \{f_1, f_2, \ldots, f_m\}$ such that $f_1 < f_2 < \ldots < f_m$, and for all $\pi_j \in \Pi_y$, curr_freq[y] $= m$. In Figure 5.13b, for all $\pi_j \in \Pi_y$, the initial value of curr_freq[y] = 2 denoting that the highest frequency level is $f_2 \in F$. For all $\pi_j \in \Pi_y$, when action jump_21[y] is taken, the value of curr_freq[y] changes to 1.

Priced Timed Automaton Scheduler. The automaton *Scheduler* models the scheduler, defined as,

$$(L, l^0, Act, E, P, Inv)$$

All components of *Scheduler* model are defined in the following.

- The location $L = l_0 = \{\mathsf{Initial}\}$ is the only location in *Scheduler* model.
- For $F = \{f_1, \ldots, f_k, f_i\}$ such that $f_1 < f_2 < \ldots < f_k < f_i$, the action set $Act = \{\mathsf{jump\_12}, \ldots, \mathsf{jump\_ik}\}$ parametrised with the VFI ids, synchronises with the PTA $Processor_1, \ldots, Processor_n$.
- For $F = \{f_1, \ldots, f_k, f_i\}$ such that $f_1 < f_2 < \ldots < f_k < f_i$, and $\pi_j \in \Pi_y$, the edge set E has the following edges for broadcast such that:
  - $\mathsf{Initial} \xrightarrow{\mathsf{freq\_lev}[y]==0 \wedge \mathsf{curr\_freq}[y]==i:\, \mathsf{jump\_ik}[y]!,\, \varnothing} \mathsf{Initial}$,
  - $\mathsf{Initial} \xrightarrow{\mathsf{freq\_lev}[y]==0 \wedge \mathsf{curr\_freq}[y]==k:\, \mathsf{jump\_ki}[y]!\, \varnothing} \mathsf{Initial}$,
    $\vdots$

– Initial $\xrightarrow{\mathsf{freq\_lev[y]==0 \wedge curr\_freq[y]==1:\ jump\_12[y]!},\ \varnothing}$ Initial

- The automaton *Scheduler* does not contain any costs.
- We do not have any clocks and invariants in *Scheduler*. Therefore, *Inv:* $L \rightarrow B(C)$ and $Inv(l^0) = true$.

For example, in Figure 5.13c, the action jump_21[y] is enabled when all processors $\pi_j \in \Pi_y$ are in the idle state and the current frequency level is $f_2 \in F$, i.e., freq_lev[y] == 0 && curr_freq[y] == 2. When this action is taken, all processors $\pi_j \in \Pi_y$ synchronise with the automaton *Scheduler*, and change the frequency level to $f_1 \in F$. Same is the case with the other action jump_12[y].

5.7 Energy Optimisation of SDF Graphs using UPPAAL CORA

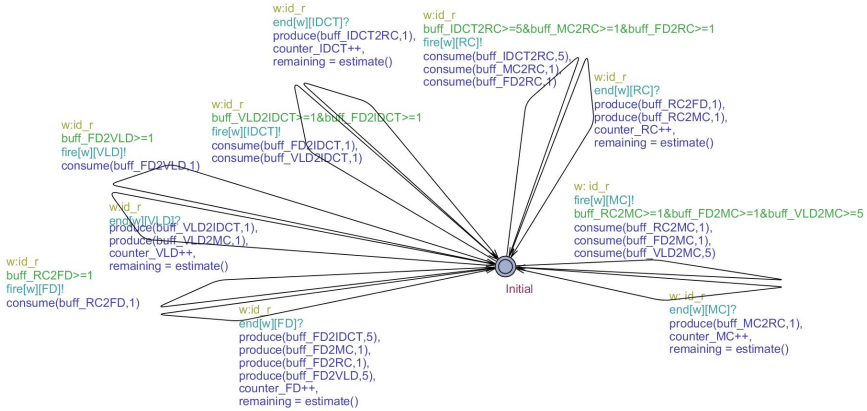
This section illustrates how we use UPPAAL CORA to obtain energy-optimal schedules. As explained earlier in Chapter 2, each actor fires according to the repetition vector γ in an iteration. For each actor $a \in A$ in the SDF graph, we define its corresponding entry in the repetition vector as $\gamma(a)$. We also define the number of iterations per period as *iter*.

A technique of calculating the maximum throughput of an SDF graph mapped on a given number of processors via timed automata (TA), using the model checker UPPAAL is proposed in Chapter 4. This work demonstrates that the *fastest execution* of every consistent and strongly connected SDF graph, mapped on a platform application model, repeats the periodic phase n times if each actor $a \in A$ fires equal to $(n \cdot iter + k) \cdot \gamma(a)$ for some constants n and k. The maximal throughput of the SDF graph is determined from the periodic phase. For example, we know that the repetition vector γ of the example SDF graph given in Section 5.4 is $\langle \mathrm{FD}, \mathrm{VLD}, \mathrm{IDC}, \mathrm{RC}, \mathrm{MC} \rangle = \langle 1, 5, 5, 1, 1 \rangle$. We can find out the throughput using the 3^{rd} multiple of the repetition vector, i.e., $(n \cdot iter + k) = 3 \cdot \gamma(a)$ for all actors $a \in A$, by selecting the *Fastest* trace option in UPPAAL, and verifying the following query.

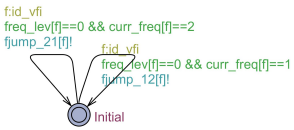
$$\mathbf{E} <> (\texttt{counter\_FD} == 3 \texttt{ and counter\_VLD} == 15 \texttt{ and counter\_IDC} == 15 \texttt{ and counter\_RC} == 3 \texttt{ and counter\_MC} == 3)$$

We find the periodic phase from the generated trace, representing the maximum throughput. To compute an energy-minimal schedule from an SDF graph G and a PAM $\mathcal{P}$, we perform the following steps.

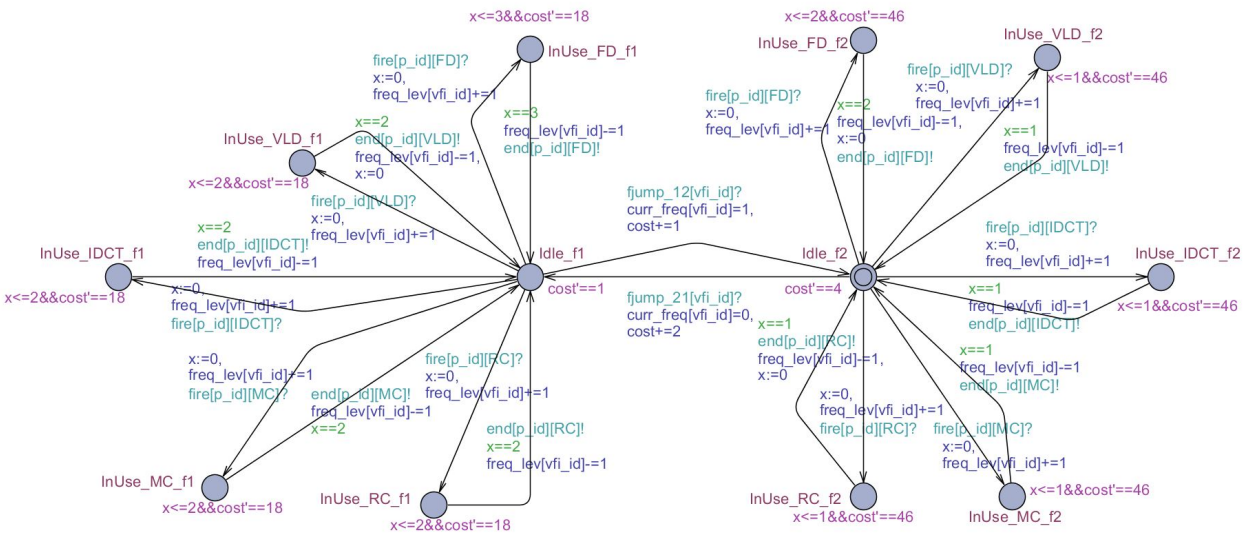
1. TA models are extracted such that $A_G \| Processor_1 \|, \ldots, \| Processor_n$.
2. We obtain the time T needed to complete the *fastest execution* of A_G, by running the query $\mathbf{Q_1} = \mathbf{E} <> (\bigwedge_{\mathtt{a} \in \mathtt{A}} \mathtt{counter\_a} = (\mathtt{n} \cdot \mathtt{iter} + \mathtt{k}) \cdot \gamma(\mathtt{a}))$ in UPPAAL.
3. PTA models are extracted such that $A_G \| Processor_1 \|, \ldots, \| Processor_n \|$ *Scheduler*.



(a) UPPAAL CORA model A_G for MPEG-4 decoder in Figure 5.2 having five actors



(b) Scheduler model used to change the frequency levels



(c) UPPAAL CORA model $Processor_j$ onto which MPEG-4 decoder in Figure 5.2 is mapped

Figure 5.13: UPPAAL CORA editor showing SDF graph, Processor and Scheduler models

4. We obtain the cheapest trace finished within time T, by running the query $\mathbf{E}<>(\mathtt{Q}_1 \wedge \mathtt{time} \leq \mathtt{T})$ in UPPAAL CORA.

5. The trace is translated into an energy-optimal schedule. That is, by considering the action labels on the transitions, we know which actor is executed on which processor at which frequency.

For example, the *fastest execution* of the query mentioned above completes in 18 ms, if the corresponding SDF graph is mapped on four processors. If we add the constraint $\mathtt{global} = 18$ to our earlier query in UPPAAL CORA, we get the optimal schedule in terms of energy utilisation at the maximum throughput on a given number of processors. The clock variable global is used to observe the overall time progress, and is never reset.

5.8 Experimental Evaluation via MPEG-4 Decoder

We analyse results of energy optimisation by means of an example of the MPEG-4 decoder example in Figure 5.2 on page 101. We evaluate energy consumption with respect to (1) fixed number of processors (2) varying number of processors. Finally, the method of verifying various user-defined properties using model checking is explained.

5.8.1 Fixed Number of Processors

We consider an MPEG-4 decoder mapped on the platform containing four Samsung Exynos 4210 processors, i.e., $\Pi = \{\pi_1, \pi_2, \pi_3, \pi_4\}$. For the constraint of finishing three graph iterations with respect to the varying deadlines, Figure 5.14 shows the energy consumption calculated for each scenario. The first two scenarios are compared as follows.

GDVFS vs GDVFS+DPM

- In almost all cases, considering DVFS only (GDVFS) results in higher energy consumption, as compared to considering the combination of DVFS and DPM (GDVFS + DPM). However, at the deadline of 30 ms, energy consumption in GDVFS + DPM surpasses GDVFS. If we compare the schedule of GDVFS and GDVFS + DPM at the deadline of 30 ms, we notice that it remains the same. However, considering GDVFS + DPM includes transition overheads incursion to move to idle states, making it less energy-optimal than GDVFS.

- At tighter deadlines when idle time of the processors is not sufficient to move to low power state, the difference between GDVFS and GDVFS + DPM is not significant. Thus, E_{Tot} lies in Region A. However, as the deadline is relaxed, the processors spend more time in low power state and E_{Tot} moves to Region B. Consequently, GDVFS + DPM gets more promising, implying the benefits of DPM. For example, at the deadline of 50 ms, GDVFS + DPM saves significant energy consumption equal to 10.3% compared to GDVFS.

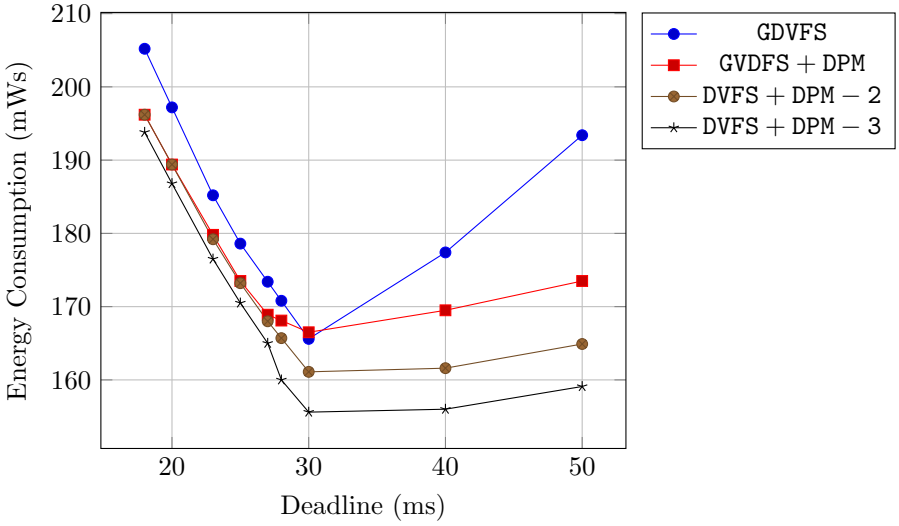


Figure 5.14: Comparison of power optimisation techniques for different scenarios

Therefore, the results explained above prove our earlier claim that static power is non-negligible in order to guarantee optimality, and both Region A and B must be analysed to determine minimum energy consumption.

Now we have seen the benefits of DPM, the effect of varying the number of VFI partitions, i.e., `GDVFS + DPM`, `DVFS + DPM − 2` and `DVFS + DPM − 3` is described below.

DVFS+DPM with VFIs

- At tighter deadlines, for the reason that the system is at the maximum capacity all the time, having higher number of VFIs does not result in major energy reduction.
- But, as the deadline is relaxed, we see that increasing the number of VFIs prove to be more effective, and produce considerable reduction in energy consumption. For example, for the deadline of 50 ms, `DVFS + DPM − 2` and `DVFS + DPM − 3` save 4.9% and 8.3% energy consumption respectively, as compared to `GDVFS + DPM`. The reason is that in `GDVFS + DPM` where we have one VFI only, all processors have to run at the same frequency, even though fewer might be required. By partitioning into more VFIs, we can cluster the processors in such a way that only the required processors run at the specific frequency, and others may run at the different frequency; thus, trading system's complexity for energy minimisation.

Hence, VFIs provide better control over energy optimisation and design complexity. Without VFIs, system designers are left with two options only, i.e., either local or global DVFS. However, with the help of VFIs, it is possible to

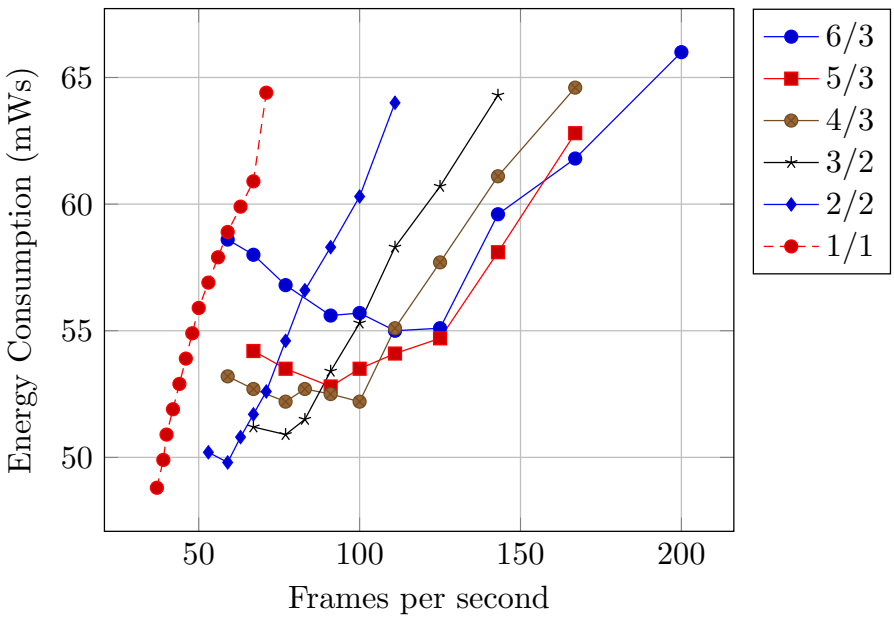


Figure 5.15: Energy usage per frame against Frames per second. The legend shows the number of processors/VFIs.

achieve fine-grained energy reduction by employing any DVFS policy, ranging from local to global. Therefore, the use of VFIs enables system designers with the larger range of design choices.

5.8.2 Varying Number of Processors

We also evaluate the performance of the MPEG-4 decoder on a varying number of processors. The maximum number of processors required for self-timed execution of this example is 6, calculated by SDF^3. We obtain a Pareto front by sweeping the throughput constraint, as shown in Figure 5.15. We get three majors results from Figure 5.15, as explained below.

- Achieving higher frames per second at fewer processors increases the energy consumption. The reason is the smaller slack at the tighter frames per second constraint. Therefore, more work is done on the fewer processors to attain same frames per second.

- As we relax the frames per second constraint, slack increases, and the same frames per second can be achieved by consuming less energy on fewer processors. For instance, in Figure 5.15, we can reach 100 frames per second on four processors with 2.4% less energy consumption, as compared to five processors. For higher slack in the application, this difference gets bigger. Thus, we may not require more processors in our platform, and

reach a certain throughput at a considerably lower energy consumption.

- Relaxing the frames per second beyond a certain limit increases the energy consumption, as the static energy surpasses the dynamic energy. For instance, the energy consumption of three processors increases by 1.9%, when moving from 77 to 59 frames per second.

5.8.3 Quantitative Analysis

We can analyse several functional and temporal properties of an MPEG-4 decoder using model checking. This includes simple reachability properties such as, "does RC eventually fire?" and "after five consecutive VLD firings, MC must fire at least once". We can also check safety properties such as, "all processors belonging to the same VFI should never run at the different frequency". Similarly, liveness properties such as, "after a processor is occupied, it is eventually released" can also be verified.

5.9 Other Case Studies

Apart from the MPEG-4 decoder example, we present other real-life case studies namely, an MP3 decoder in Figure 2.15, an MP3 playback application in Figure 2.16, and an audio echo canceller in Figure 2.17. We also used an artificial bipartite SDF graph with 4 actors in Figure 2.18. The execution times of these case studies are given in ms. We assume that these case studies are mapped on a multiprocessor platform containing Samsung Exynos 4210 processors $\Pi = \{\pi_1, \ldots, \pi_n\}$. Table 5.4 shows the considered frequency levels, and assumed power consumption of Exynos 4210 processors. For easier understanding, we only consider deadline constraint equal to minimum achievable time (ms) per iteration on a given number of processors. We also assume that, for all actors $a \in A$, $\tau_{act}(a, f_1) = \lceil \frac{\tau_{act}(a,f_2)}{0.738} \rceil$.

Table 5.16 shows the results of the experiments to find out the least power consumption. The first column displays the given number of processors, and the second column represents the division of processors into VFIs. Columns 3-4 depict per iteration, minimum achievable time (ms) and minimum energy consumption (mWs) respectively, on the given number of processors.

We could determine the exact number of processors required for a self-timed execution, using SDF^3. Then, we apply our approach to derive an optimal schedule on a smaller number of processors to determine the least energy usage. As we can see in case of a bipartite graph in Table 5.16, reducing the number of processors to three does not deteriorate minimum time per iteration considerably, and decreases slightly to 44 ms. Nonetheless, the decrease in energy consumption per iteration is significant, equal to 6.8 mWs, due to presence of higher slack in the application. It clearly shows, that similar performance with substantially less energy dissipation can be achieved, even with fewer processors than required for a self-timed execution. Thus, using model checking, we generate an energy-optimal schedule automatically in a simple manner, on a given number of processors partitioned into VFIs, once the target state is specified in a query.

Number of Processors	VFIs	Time per Iteration	Energy Consumption
MP3 Decoder in Figure 2.15			
2	$\Pi_1 = \{\pi_1, \pi_2\}$	8	64·6
1	$\Pi_1 = \{\pi_1\}$	14	64·4
MP3 Playback Application in Figure 2.16			
2	$\Pi_1 = \{\pi_1, \pi_2\}$	1880	9907
1	$\Pi_1 = \{\pi_1\}$	2118	9742·8
Audio Echo Canceller in Figure 2.17			
4	$\Pi_1 = \{\pi_1, \pi_2\}, \Pi_2 = \{\pi_3, \pi_4\}$	23	324·2
3	$\Pi_1 = \{\pi_1\}, \Pi_2 = \{\pi_2, \pi_3\}$	24	322·3
2	$\Pi_1 = \{\pi_1, \pi_2\}$	35	322
1	$\Pi_1 = \{\pi_1\}$	73	335·8
Bipartite Graph in Figure 2.18			
4	$\Pi_1 = \{\pi_1, \pi_2\}, \Pi_2 = \{\pi_3, \pi_4\}$	42	345·3
3	$\Pi_1 = \{\pi_1\}, \Pi_2 = \{\pi_2, \pi_3\}$	44	338·5
2	$\Pi_1 = \{\pi_1\}, \Pi_2 = \{\pi_2\}$	51	333·1
1	$\Pi_1 = \{\pi_1\}$	73	335·8

Table 5.16: Energy optimisation results for various case studies calculated using UPPAAL CORA

So far, we have assumed a homogeneous system in which an actor can be mapped on any processor. A homogeneous system gives more freedom to decide which actor to assign to a particular processor. However, this freedom is limited in a heterogeneous system by which processors could be utilised to execute a particular actor.

In UPPAAL CORA, we can utilise the same models described earlier in a heterogeneous system. Let us consider the SDF graph of the MPEG-4 decoder mapped on a heterogeneous system containing two Samsung Exynos 4210 processors $\Pi' = \{\pi'_1, \pi'_2\}$ and two Samsung Exynos 4212 [SAMb] processors $\Pi'' = \{\pi''_1, \pi''_2\}$. We assume that both Exynos 4210 and 4212 processors are available with the same frequency levels (MHz) $\{f_1, f_2\} \in F$ such that $f_2 = 1400$ and $f_1 = 1032.7$. Furthermore, let us consider the following assumptions also.

$$P_{tr}(\pi', f_2, f_1) = P_{tr}(\pi'', f_2, f_1)$$
$$P_{tr}(\pi', f_1, f_2) = P_{tr}(\pi'', f_1, f_2)$$

For all $\pi' \in \Pi', \pi'' \in \Pi''$ and $f \in F$,

$$P_{idle}(\pi', f) = P_{idle}(\pi'', f)$$
$$P_{occ}(\pi', f) = P_{occ}(\pi'', f)$$

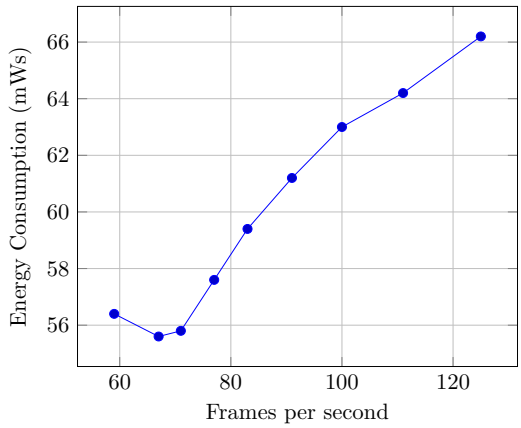


Figure 5.17: Optimal energy consumption of MPEG-4 decoder in Figure 5.2 in a heterogeneous system

Let us consider that the platform is implemented in such a way that actor $\{\text{FD}\} \subseteq A$ can be mapped only on the processor $\{\pi'_1\} \subseteq \Pi'$, actors $\{\text{VLD, IDC}\} \subseteq A$ can be executed only on the processors $\{\pi'_2, \pi''_2\} \subseteq \Pi' \cup \Pi''$, and the processor $\{\pi''_1\} \subseteq \Pi''$ is assigned to execute actors $\{\text{RC, MC}\} \subseteq A$ only. The processors are partitioned into VFIs in such a way that, $\Pi_1 = \{\pi'_1, \pi''_1\}$ and $\Pi_2 = \{\pi'_2, \pi''_2\}$. Figure 5.17 shows the Pareto front of total energy consumption for varying throughput constraint.

5.10 Tool Support

To automate the energy-optimal scheduling of SDF graphs, a tool-chain termed COMET (**CO**-design using **M**odel-Driven **E**ngineering for Da**T**aflow Applications) is developed based on the principles of Model-Driven Engineering (MDE). Figure 5.18 shows the workflow of the COMET tool-chain. It takes as an input an SDF graph using SDF^3, and a PAM. Then, it transforms these components to UPPAAL CORA, which in turn outputs optimal schedules. Later, Chapter 7 provides a detailed explanation of benefits of using MDE.

5.10.1 Input

The input to the COMET tool-chain is an SDF graph developed using SDF^3, and a PAM. For easier modelling of PAMs, a domain-specific visual editor is built using EuGENia [KRPP09]. Figure 5.19 shows the screen shot of an example PAM containing four Exynos 4210 processors partitioned into two VFIs. Each processor is equipped with two frequency levels. The top part of the figure shows the processors and VFIs. The bottom part of the figure shows the frequency

All components of the COMET tool-chain can be found at `https://github.com/utwente-fmt/COMET`. An instruction manual to use the COMET tool-chain is also given in this repository.

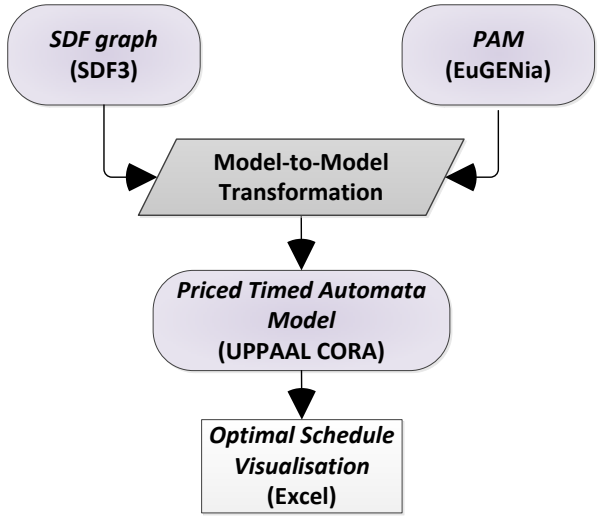


Figure 5.18: The workflow of the COMET tool-chain. After the SDF^3 model and PAM are transformed to UPPAAL CORA using the model-to-model transformation, the optimal schedules are generated.

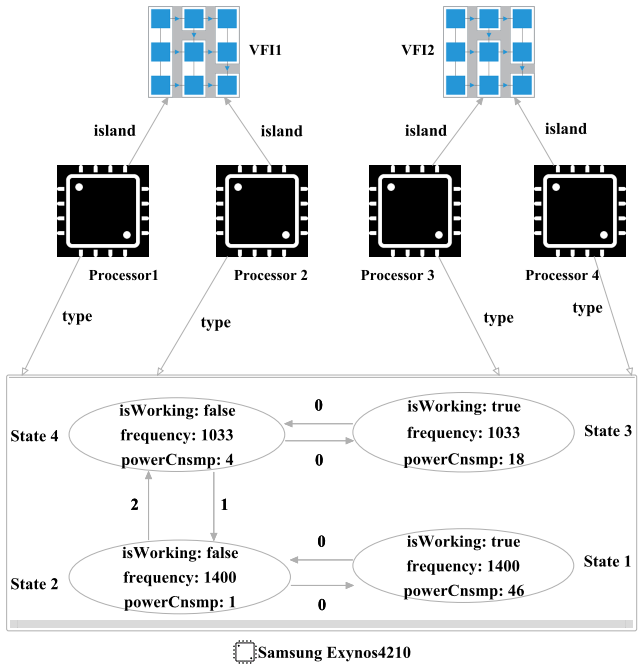


Figure 5.19: PAM created using visual editor

Time	Processor1	Processor2	Processor3	Processor4
0	Idle_f2	Idle_f2	Idle_f2	FD_f2
2	VLD_f2	VLD_f2	Idle_f2	VLD_f2
3	VLD_f1	IDC_f1	IDC_f1	VLD_f1
5	IDC_f1	IDC_f1	IDC_f1	MC_f1
7	Idle_f2	RC_f2	Idle_f2	Idle_f2
8	Idle_f2	Idle_f2	Idle_f2	Idle_f2

Figure 5.20: Example schedule of the MPEG-4 decoder on four processors. The columns show the processors, the rows show the time, and the cells represent the name of the actor and the frequency level at which they are fired. If there is no actor being fired, then the cells represent at which frequency level, a processor is idle.

levels and transition overheads. For example, at 1400 MHz, a processor consumes 0.1 W if it is idle, or 4.6 W if it is working. Furthermore, a processor consumes 0.2 W if changes its frequency from 1044 MHz to 1033 MHz.

5.10.2 Transforming Models to UPPAAL CORA Models

After creating the SDF^3 models and PAMs, the next step is to transform these models automatically to PTA models in UPPAAL CORA format. This is done using a model-to-model transformation developed in *Epsilon Transformation Language* (*ETL*) [KPP08]. The generated PTA models are already explained in Section 5.6.

5.10.3 Output

The problem of finding energy-optimal schedules (while satisfying minimal throughput requirements) is given as an optimisation problem, defined as a reachability property over PTA models. The property is then checked by UPPAAL CORA which extracts an energy-optimal trace. The trace is translated to a schedule in a Excel sheet showing which actor at which frequency level (cells in the Excel sheet) is fired on which processor (columns in the Excel sheet) at what time (rows in the Excel sheet). Figure 5.20 shows an example schedule of the MPEG-4 decoder in Figure 5.2 on page 101 on four processors. For example, at time t = 0 ms, the actor *FD* is fired on the processor π_4 at the frequency level f_2. If there is no actor being fired, then the cells represent at which frequency level, a processor is idle. For example, at time t = 0 ms, the processor π_1 is idle at the frequency level f_2.

5.11 Conclusions

Despite the remarkable progress in energy optimisation of deadline-constrained applications, compact methods for optimal energy management of SDF graphs are still needed. In addition, with the growth of processing power in battery-constrained devices, efforts must be made to keep energy utilisation to minimum.

We have demonstrated a novel energy reduction technique for SDF-modelled streaming applications, which combines the benefits of DVFS and DPM using model checking. This technique can be applied to any multiprocessor heterogeneous platform, having transition overheads and partitions of VFIs.

CHAPTER 6

Model Checking and Evaluating QoS of Batteries

Abstract

System lifetime is a major design constraint for battery-powered mobile embedded systems. The increasing gap between the energy demand of portable devices and their battery capacities is further limiting system lifetime of mobile devices. Thus, the guarantees over the Quality of Service (QoS) of the battery-constrained devices under strict battery capacities are of primary interest for mobile embedded systems' manufacturers and stakeholders.

In earlier chapters 4 and 5, we focused on resource and energy-optimal scheduling of SDF graphs, but considered an ideal battery source. In this chapter, we extend the work in Chapter 5 by an intuitive battery model termed kinetic battery models (KiBaMs). In this way, processors are dependent on the charge from these batteries to run. Once the batteries are empty, the processors cannot run any more. This signifies the end of the *system lifetime*. We model the whole system including SDF graphs, heterogeneous multiprocessor platforms, and multiple kinetic battery models (KiBaMs) as hybrid automata. Afterwards, we apply Monte Carlo simulations to evaluate, (1) system lifetime; and (2) minimum required initial battery capacities to achieve the desired application performance. We demonstrate that our approach shows a significant improvement in terms of scalability, as compared to a priced timed automata based KiBaM approach. This approach also allows early detection of design errors via model checking.

About this chapter: The current chapter is based on the paper "Model Checking and Evaluating QoS of Batteries in MPSoC Dataflow Applications via Hybrid Automata", which was published at ACSD 2016 [AJSvdP16a]. An extended report on the work was published at University of Twente Eprints [AJSvdP16b]. The original paper largely remains the same.

6.1 Introduction

Mobile computing has experienced a major upswing over last two decades. As a result, applications with increasing functionality and complexity are continuously implemented on mobile embedded devices such as smart phones and satellites, allowing these systems to operate independently. For example, modern-day satellites are capable of transmitting videos, communicating with aeroplanes, providing navigation to automobiles etc., compared to the first-generation satellites which were only able to transmit radio signals. However, this trend also has increased the energy consumption of mobile devices manifold. On the other hand, battery energy densities have not grown at the same rate over the years, thus leading to system lifetime as a major design constraint [Cha07]. We define the lifetime as the time one can use the battery before it is empty.

Mobile embedded systems are often powered only by batteries that may or may not be recharged regularly by an external power source. For example, in a military Software Defined Radio that is being operated in a desert or on a mountain where energy supplies are unreliable, the primary Quality of Service (QoS) concern is to determine system lifetime. Also, a geostationary satellite with solar panels to charge on-board batteries, is recharged at a regular intervals of 12 hours when facing the sun. However, the satellites have strict limitations regarding mass and volume. In this case, the main QoS interest is to assess the battery sizes and weight that yield the relevant performance criteria. To summarise, the evaluation of the QoS of battery-constrained mobile embedded systems has emerged as one of the most critical, challenging and essential concern for manufacturers, investors and users of such systems.

One can identify three QoS factors, and their relation with different design choices, as given in Table 6.1. First, the *throughput* of a system, defined as a measure of how many units of information a system can process in a given amount of time, has a direct impact on system lifetime. Secondly, the number of processors affects both system lifetime, and manufacturing cost of the overall system. Lastly, the number of batteries relates not only to system lifetime and cost, but also to mass and volume of a system. Therefore, this chapter takes in account aforementioned design alternatives, with respect to system lifetime and minimum batteries' capacities.

We consider a very intuitive battery model termed Kinetic Battery Model (KiBaM) [MM93] as a representation of dynamic behaviour of a conventional rechargeable battery, see Figure 6.4 on page 133. A KiBaM models the total charge in a battery as two separate tanks separated by conductance. One tank holds the charge which is immediately available to be consumed by the load. The other tank holds the charge which is chemically bound. For a given load current, a KiBaM describes the charge stored in a battery by two coupled differential equations. Experimental studies show that the KiBaM provides a good approximation of system lifetime across various battery types [JH09].

Design Choices \ QoS Factors	System Lifetime	Cost	Volume and Mass
Throughput	✓		
Number of Processors	✓	✓	
Number of Batteries	✓	✓	✓

Table 6.1: Relation between design choices and QoS factors. ✓represents that the QoS factor holds for the given design choice.

6.2 Background

In this section, we explain two important factors that affect the system lifetime, (1) load level, and (2) scheduling pattern of the batteries. We also introduce a novel method from the literature [JHBK09] that generates the optimal battery schedules, and discuss its limitations.

Load level of batteries. The level of the load current can be altered using the power management techniques already introduced in Chapter 5, namely Dynamic Power Management (switching to low power state) (DPM) [BBDM00] and Dynamic Voltage and Frequency Scaling (throttling processor frequency) (DVFS) [WWDS94]. The concept of voltage-frequency islands (VFIs) [HM07] further allows us to cluster a group of processors in such a way that each VFI runs on a common clock frequency/voltage. Furthermore, different VFI partitions represent DVFS *policies* of different granularity. Thus, with the help of VFIs, we can combine DPM, and DVFS policy with any granularity, generalising local and global DVFS. To further illustrate the relation of power management in processors, and system lifetime, let us consider an example below.

A typical system configuration for connecting a battery to a voltage/frequency scalable processor is shown in Figure 6.2. The battery's voltage and current is represented by V_{bat} and I_{bat}, and the processor's voltage and current is represented by V_{proc} and I_{proc}. Portable electronic devices, such as, mobile phones, satellites, and laptop computers often contain several sub-circuits, each with its own voltage level requirement, that is different from the voltage supplied by the battery. Hence, a DC-DC converter is utilised to convert DC (direct current) power provided by the battery from one voltage level to another. If we represent the efficiency of the DC-DC converter by η, the voltage/frequency scaling is governed by the following equation.

$$\eta \times V_{bat} \times I_{batt} = V_{proc} \times I_{proc} \tag{6.1}$$

Modern day microprocessors are designed using a specific circuitry design technology, termed as complementary metal-oxide semiconductor (CMOS). In CMOS

Figure 6.2: System level configuration of a single processor. V_{batt} (I_{bat}) and V_{proc} (I_{proc}) represents the voltage (current) of the battery and processor respectively.

based processors, voltage/frequency scaling by a factor of s causes the processor current I_{proc} and the battery current I_{batt} to scale by a factor of s^2 and s^3 respectively [CC02]. Therefore, slack utilisation by DVFS and DPM can greatly affect the load current, which in turn can impact the overall system lifetime. Moreover, partitioning processors into VFIs provide even better control over system lifetime. Without VFIs, systems are left with two options only, i.e., achieving I_{batt} with respect to either local or global frequency, resulting in unoptimised system lifetime. However, with the help of VFIs, it is possible to prolong the system lifetime, by modifying I_{batt} with respect to any frequency, ranging from local to global. We have shown in Chapter 5 that only the combination of DPM and DVFS, and partitioning of processors into VFIs guarantees energy optimisation.

Scheduling of batteries. If we have multiple batteries in a system, another important factor contributing to the overall lifetime is the usage pattern of batteries, i.e., how batteries are scheduled. This leads to an important research problem of devising a battery-aware scheduling mechanism, where given a set of tasks, a set of resources to execute the tasks, and a given number of multiple batteries, we are able to derive a battery-optimal schedule of tasks. However, the charge stored in a battery is represented by a finite set of continuous variables in a KiBaM, making the behaviour of the KiBaM hybrid. Evaluating the performance of various (battery-) scheduling strategies using existing analysis techniques for hybrid systems, is very expensive [WHL14].

To address this problem, the state-of-the-art method in [JHBK09] discretises the KiBaM, and models it as priced timed automata (PTA). Furthermore, for a fixed execution order of tasks, this approach deploys the model checker UPPAAL CORA that searches the whole state-space and generates the optimal battery schedule, using the well-developed model-checking techniques for PTA. However, this method also does not solve the scalability problem. As increasing the initial battery capacities leads to searching the bigger state-space, this approach only allows to model limited total battery capacities.

6.3 Methodology and Contributions

We propose an approach based on Hybrid Automata (HA) [HR98] introduced in Chapter 3. In contrast to discretisation, as done in [JHBK09], we take into account the continuous variables of the KiBaM by modelling it as a hybrid automaton, which obviously makes it a more accurate model. This approach

enables us to utilise UPPAAL SMC to employ the highly scalable technique of Monte Carlo simulations to assess various QoS parameters, such as system lifetime and adequate battery capacities. We show that our approach scales better than the one presented in [JHBK09]. Furthermore, we utilise UPPAAL SMC also for applying model checking to verify various user-defined properties. Thus, as opposed to other simulation based tools for hybrid systems, modelling as HA and using UPPAAL SMC provides an additional benefit of model checking.

Our approach takes four ingredients: (1) a platform model that describes the specifics of the hardware, such as, VFI partitions, frequency levels and power usage per processor; (2) an SDF graph scheduler that maps application tasks on the platform model in a static-order manner; (3) given number of batteries; and (4) a battery scheduler that defines the scheduling scheme. For given battery capacities and timing constraints, we compute system lifetime (SDF graph iterations). Similarly, for given application performance criteria, we determine adequate battery capacities. This method facilitates system designers to evaluate aforementioned QoS factors for different design choices, such as, varying number of VFIs, processors, and batteries. Furthermore, this method also allows system designers to detect subtle battery design errors in early phases via model checking. In particular, our main contributions are as follows.

- Assessing QoS of multiple KiBaMs for different design alternatives, without discretising time.
- We consider realistic hardware platforms equipped with the novel energy management techniques, compared to the state-of-the-art [WHL14];
- We show that our approach allows better scalability than PTA-based discretised KiBaM [JHBK09];
- Our approach allows early detection of design errors via model checking.

Chapter Outline. Section 6.4 reviews related work. Section 6.5 provides all relevant definitions, and translation to HA using UPPAAL SMC is illustrated in Section 6.6. Section 6.7 experimentally evaluates the QoS analysis, and Section 6.8 verifies the functional and temporal properties using model checking. Finally, Section 6.9 draws conclusions and outlines possible future research.

6.4 Related Work

An extensive survey paper [JH09] outlines the broad research work on various battery models. The state-of-the-art methods in the realm of battery-aware scheduling for multiple batteries, are presented in [JHBK09] and [FLM11]. The approach in [JHBK09], in comparison to ours, discretises time. This approach helps to find optimal battery schedules, but do not scale well because of discretisation. The technique in [FLM11] models KiBaMs as hybrid like us, and discretises time to search the state-space, leading to the better results than the work in [JHBK09]. But, due to the fact that the state-space grows larger with the number of batteries, the scalability of this approach also suffers. We,

Method	Without Discretisation	Multiple KiBaMs
[JHBK09, FLM11]	✗	✓
[WHL14]	✓	✗
[HKN15]	✓	✗
Our Method	✓	✓

Table 6.3: Comparison of different KiBaM analysis methods

on the other hand, run Monte Carlo simulations, that allows us to avoid the state-space explosion. The analysis shows that the scalability of our approach is better than the technique in [JHBK09].

A more advanced technique that utilises hybrid automata like us, is presented in [WHL14]. In this paper, the KiBaM provides energy to an uniprocessor. Unlike our method, this approach discusses a single battery case only. Another novel work in [HKN15] extends KiBaMs with random initial SoC and load, without discretising time. In this way, probabilistic guarantees about the system lifetime can be provided. In comparison to our work, this technique is also confined to a single KiBaM only. Table 6.3 summarises different aforementioned KiBaM analysis methods.

6.5 System Model Definition

In this section, we first explain KiBaMs and define a KiBaM system. Then, we recall the definition of SDF graphs from Chapter 2. In this chapter, we consider that the processors depend on the charge available in the batteries. Therefore, we redefine the platform application model (PAM) introduced in Chapter 5, and replace the concept of *power consumption* with *charge consumption*. The execution time of actors is dependent on the running frequency of processors in a PAM. Therefore similar to Chapter 5, the execution time of actors are included in the definition of PAMs instead of SDF graphs.

6.5.1 Kinetic Battery Model

The kinetic battery model (KiBaM) [MM93] is a mathematical characterisation of state of charge of a battery. Key feature is that not all energy stored in a battery can be utilised at all times. To model this phenomenon, the total charge stored in a battery is divided into two "tanks" respectively termed as, the *available charge* and the *bound charge*, see Figure 6.4. Only the available charge can be consumed immediately by a *load* at the time-dependent rate i, and thereby behaves similar to an ideal energy source. During low or no discharge current, some of the bound charge is converted to available charge. This conversion is at a rate proportional to the height difference with the proportionality factor being the rate constant k, and is available to be consumed. Thus, the *available charge*

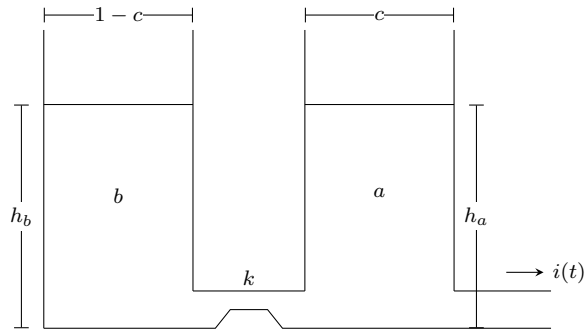


Figure 6.4: Model of a KiBaM showing available and bound charge tanks

replenishes *bound charge*, and this effect is termed as *recovery effect*.

If the widths of the available and bound charge tanks are given by c and $1 - c$ respectively, then the tanks are filled to heights h_a and h_b, and the charges in both tanks are $a = ch_a$ and $b = (1 - c)h_b$ respectively. Formally, a KiBaM is characterised by the following system of differential equations.

$$\dot{a}(t) = -i(t) + k(h_b - h_a) \tag{6.2}$$

$$\dot{b}(t) = -k(h_b - h_a) \tag{6.3}$$

The system starts in an equilibrium, i.e. $h_a = h_b$. With an initial capacity of C, the initial conditions are $a(0) = cC$ and $b(0) = (1 - c)C$. The battery is considered empty when $a = h_a = 0$, as it cannot supply charge any more at the given moment even though it may still contain bound charge. In fact, due to the dynamics of the system, the bound charge cannot reach zero in finite time. The system lifetime ends when all batteries are emptied.

The differential equations can be solved using Laplace transformation, which gives:

$$y_1 = y_{1,0}e^{-k't} + \frac{(y_0k'c - i)(1 - e^{-k't})}{k'} - \frac{ic(k't - 1 + e^{-k't})}{k'} \tag{6.4}$$

$$y_2 = y_{2,0}e^{-k't} + y_0(1 - c)(1 - e^{-k't}) - \frac{i(1 - c)(k't - 1 + e^{-k't})}{k} \tag{6.5}$$

where k' is defined as:

$$k' = \frac{k}{c(1 - c)}, \tag{6.6}$$

and $y_{1,0}$ and $y_{2,0}$ are the amount of available and bound charge, respectively, at $t = 0$. For y_0, we have $y_0 = y_{1,0} + y_{2,0}$.

Definition 6.1. *A* KiBaM system *is a tuple* $\mathcal{KS} = (B, Cap)$ *consisting of,*

- *a finite set of KiBaMs* $B = \{bat_1, \ldots, bat_m\}$*, and*
- *a function* $Cap : B \to \mathbb{R}_{\geq 0}$ *denoting the initial capacity of a KiBaM* $bat \in B$*.*

As discussed earlier, in case of more than one battery in a system, the batteries are chosen according to some schedule or scheduling policy. In most systems, the batteries are used sequentially, i.e., only when one battery is empty, the other is used [JHBK09]. However, by switching between batteries, their recovery effect is utilised, which in turn extends the overall system lifetime [JHBK09]. We consider a scheduling scheme termed *round-robin*. In this scheduling scheme, after an SDF graph iteration finishes, (i.e., not during the execution of the iteration) the next available battery in a circular order is selected to provide energy for the next iteration.

Example 6.2. Let us consider that we have two KiBaMs $B = \{bat_1, bat_2\}$ in our KiBaM system. The supposed capacity of both batteries is, $Cap(bat_1) = Cap(bat_2) = 50$ mAs. As mentioned earlier, we consider *round-robin* scheduling scheme, in which a new task is served by the next available battery in a fixed order.

Figure 6.5 shows the simulation of an arbitrary periodic load. The (solid) red and (dashed) blue lines represent the current load (mA) of the batteries $bat_1 \in B$ and $bat_2 \in B$ respectively. The upper solid lines represent the total charge in both batteries, i.e., the sum of the bound charge (b, not shown) and the available charge (a, the lower solid lines). Initially, the battery $bat_1 \in B$ serves the first task. When this task is going on, the available charge a_1 of the battery $bat_1 \in B$ reduces. After the current task finishes, the next task is served by the next battery, i.e., $bat_2 \in B$. In the meanwhile, bat_1 recovers, and so on. Just after time 87 and 96 ms, the available charge of both batteries expires respectively, representing the end of the system lifetime, finishing 8 tasks in total. Please note that when the system lifetime ends, both batteries have only expended about 60% its total charge. For better visibility, the load currents are scaled by multiplying by 25.

In our case studies, we consider the batteries having the capacity of 1300 mAh, as used in the Samsung Galaxy Fame smartphones [SAMc].

6.5.2 SDF Graphs

Definition 6.3. *An* SDF graph *is a tuple* $G = (A, D, \mathsf{Tok_0})$ *where:*

- A *is a finite set of actors,*
- D *is a finite set of dependency channels* $D \subseteq A^2 \times \mathbb{N}^2$*, and*
- $\mathsf{Tok_0} : D \to \mathbb{N}$ *denotes distribution of initial tokens in each channel, and*

The sets of input channel $In(a)$ *and* output channel $Out(a)$ *of an actor* $a \in A$ *are defined as:* $In(a) = \{(a', a, p, q) \in D | a' \in A \wedge p, q \in \mathbb{N}\}$ *and* $Out(a) = \{(a, b, p, q) \in D | b \in A \wedge p, q \in \mathbb{N}\}$. *The* consumption rate $CR(e)$ *and* production rate $PR(e)$ *of an channel* $e = (a, b, p, q) \in D$ *are defined as:* $CR(e) = q$ *and* $PR(e) = p$.

Informally, for all actors $a \in A$, if the number of tokens on every input channel $(a'_i, a, p_i, q_i) \in In(a)$ is greater than or equal to q_i, actor a fires and

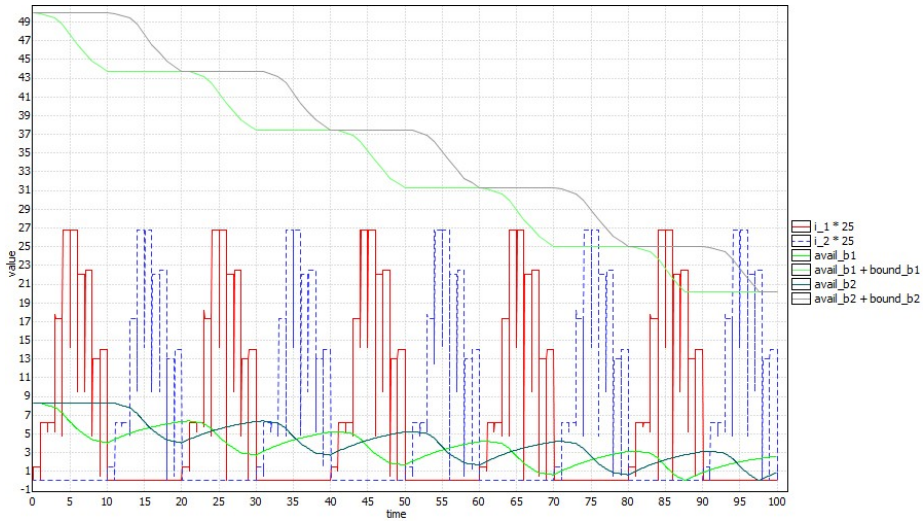


Figure 6.5: Simulation of a periodic load powered by two batteries bat_1 and bat_2. For bat_j, i_j shows the load current, *avail_bj* shows the available charge, and avail_bj+bound_bj shows the total charge. The load currents are multiplied by 25 for better visibility.

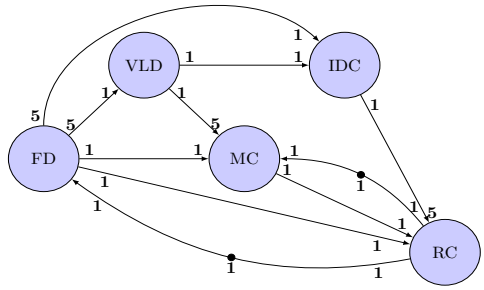


Figure 6.6: SDF graph of an MPEG-4 decoder

removes q_i tokens from every $In(a)$. The firing ends by producing p_i tokens on all $(a, b_i, p_i, q_i) \in Out(a)$.

Example 6.4. Figure 6.6 recalls the SDF graph of an MPEG-4 decoder from Chapter 2. The actors A={FD, VLD, IDC,RC, MC} represent individual tasks performed in MPEG-4 decoding. For example, the frame detector (FD) models the part of the application that determines the frame type and the number of macro blocks to decode. In our case, MPEG-4 can process only P-frames. The rest of the steps in MPEG-4 decoding are Variable Length Decoding (VLD), Inverse Discrete Cosine (IDC) Transformation, Motion Compensation (MC), and Reconstruction (RC) of the final video picture.

6.5.3 Platform Application Model

A Platform Application Model (PAM) models a multiprocessor platform where the application, modelled as an SDF graph, is mapped on. Our PAM models supports several features, including

- heterogeneity, i.e., actors can run on certain type of processors only,
- a partitioning of processors in voltage and frequency islands,
- different frequency levels each processor can run on,
- power consumed by a processor in a certain frequency, both when in use and when idle, and
- power and time-overhead required to switch between frequency levels.

Definition 6.5. *A* platform application model *(PAM) is a tuple* $\mathcal{P} = (\Pi, \zeta, F, I_{occ}, I_{idle}, I_{tr}, T_{tr}, \tau_{act})$ *consisting of,*

- *a finite set of processors* Π *assuming that* $\Pi = \{\pi_1, \ldots, \pi_n\}$ *is partitioned into disjoint blocks* $\Pi_1, \ldots, \Pi_k$ *of voltage/frequency islands (VFIs) such that* $\bigcup \Pi_i = \Pi$*, and* $\Pi_i \cap \Pi_j = \emptyset$ *for* $i \neq j$*,*
- *a function* $\zeta : \Pi \to 2^A$ *indicating which processors can handle which actors.*
- *a finite set of discrete frequency levels available to all processors denoted by* $F = \{f_1, \ldots, f_m\}$ *such that* $f_1 < f_2 < \ldots < f_m$*,*
- *a function* $I_{occ} : \Pi \times F \to \mathbb{N}$ *denoting the operating load current, if the processor* $\pi \in \Pi$ *is running at frequency level* $f \in F$ *in the working state,*
- *a function* $I_{idle} : \Pi \times F \to \mathbb{N}$ *denoting the idle load current, if the processor* $\pi \in \Pi$ *is running at a certain frequency level* $f \in F$ *in the idle state,*
- *a partial function* $I_{tr} : \Pi \times F^2 \nrightarrow \mathbb{N}$ *expressing the transition load current, in case of a frequency change by the processor* $\pi \in \Pi$ *from one frequency level* $f \in F$ *to next frequency level* $f \in F$*,*
- *a partial function* $T_{tr} : \Pi \times F^2 \nrightarrow \mathbb{N}$ *expressing the time overhead in case of a frequency change by the processor* $\pi \in \Pi$ *from one frequency level* $f \in F$ *to next frequency level* $f \in F$*, and*
- *the valuation* $\tau_{act} : A \times F \to \mathbb{N}_{\geq 1}$ *defining the execution time* τ_{act} *of each actor* $a \in A$ *mapped on a processor at a certain frequency level* $f \in F$*. For instance,* $\tau_{act}(a_i, f) = n$ *means that the actor* a_i *has an execution time* n*, if run on the frequency level* f*.*

Example 6.6. Exynos 4210 is a state-of-the-art processor used in high-end platforms such as Samsung Galaxy Note, SII etc. Table 6.7 shows its different DVFS levels, and corresponding CPU voltage (V) and clock frequency (MHz) [PPS+13].

Level	Voltage	Frequency	Level	Voltage	Frequency
1	1.2	1400	4	1.05	1128.7
2	1.15	1312.2	5	1.00	1032.7
3	1.10	1221.8			

Table 6.7: DVFS levels and corresponding CPU voltage and clock frequencies of Samsung Exynos 4210 processors

Processor	VFI	Voltage(V)	Frequency(MHz)	$\mathbf{I_{idle}}$(mA)	$\mathbf{I_{occ}}$(mA)
π_1	Π_1	1.2	$f_2 = 1400$	20	500
		1.00	$f_1 = 1032.7$	8	190
π_2	Π_2	1.2	$f_2 = 1400$	20	500
		1.00	$f_1 = 1032.7$	8	190
π_3	Π_2	1.2	$f_2 = 1400$	20	500
		1.00	$f_1 = 1032.7$	8	190
π_4	Π_3	1.2	$f_2 = 1400$	20	500
		1.00	$f_1 = 1032.7$	8	190

Table 6.8: Description of platform containing four Samsung Exynos 4210 processors π_1, π_2, π_3, and π_4. These processors are partitioned into three VFIs Π_1, Π_2, and Π_3.

Let us assume that the processors $\Pi = \{\pi_1, \pi_2, \pi_3, \pi_4\}$ are partitioned in three VFIs such that $\Pi_1 = \{\pi_1\}$, $\Pi_2 = \{\pi_2, \pi_3\}$ and $\Pi_3 = \{\pi_4\}$. Two DVFS levels (MHz) $\{f_1, f_2\} \in F$ taken from Table 6.7, i.e., $f_2 = 1400$ and $f_1 = 1032.7$, are available to all processors. The supposed transition overhead (ms) of all Exynos 4210 processors is, $\mathsf{Tr}(\pi, \mathsf{f_2}, \mathsf{f_1}) = \mathsf{Tr}(\pi, \mathsf{f_1}, \mathsf{f_2}) = 1$. Table 6.8 shows the formation of VFIs and experimental load current against each frequency level.

Definition 6.7. *Given an SDF graph $G = (A, D, \mathsf{Tok_0}, \tau)$, a* static-order *(SO) schedule is a function $\xi : \Pi \times \mathbb{R} \rightarrow (A \times F) \cup (\perp \times F) \cup (F \times F)$ that assigns to each processor $\pi \in \Pi$ over time, an ordered list of actors or idle slots to be executed at some frequency, or frequency transitions. Here $\perp$ represents the idle slots.*

Example 6.8. Table 6.9 shows an example static-order (SO) schedule. Here, $(f_i \rightarrow f_k)$ represents the frequency transition from $f_i \in F$ to $f_k \in F$. The execution of an actor $a \in A$ at a frequency level $f_i \in F$ is represented by $(a\text{-}f_i)^{ex}$, where ex indicates the consecutive executions of the actor a. Similarly, $(Idle\text{-}f_i)^{ex}$ denotes the idle time spent by a processor $\pi \in \Pi$ at a frequency level $f_i \in F$, where ex represents the duration of the idle time (ms). We assume that the execution times (ms) of all actors $a \in A$ at frequency level f_1 are rounded

π_1	π_2	π_3	π_4
$(f_2 \to f_1)$(FD-f_1)(VLD-f_1) $(f_1 \to f_2)$(IDC-f_2)^2(RC-f_2)	(Idle-f_2)^3$(f_2 \to f_1)$(VLD-f_1) (MC-f_1)$(f_1 \to f_2)$(Idle-f_2)	(Idle-f_2)^3$(f_2 \to f_1)$(VLD-f_1) (MC-f_1)$(f_1 \to f_2)$(Idle-f_2)	(Idle-f_2)^3(VLD-f_2)^2 (IDC-f_2)^2(Idle-f_2)^3

Table 6.9: Example SO schedule on four processors π_1, π_2, π_3, and π_4

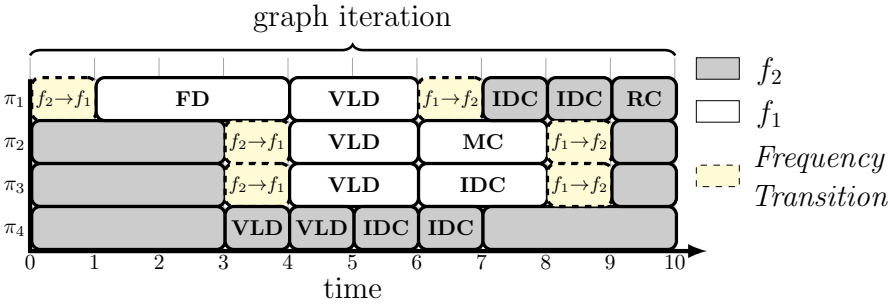


Figure 6.10: Gantt chart of the SO schedule in Table 6.9. The grey and white coloured boxes denote, if a processor is running at the frequency f_2 or f_1 respectively. Moreover, the yellow dashed boxes represent the frequency transitions.

to the next integer. As $f_1 = 0.738 \times f_2$, we obtain $\tau_{act}(a, f_1) = \lceil \frac{\tau_{act}(a,f_2)}{0.738} \rceil$. For simplicity, we omit the time of occurrence of tasks in the SO schedule, and assume that they happen consecutively.

Figure 6.10 shows the Gantt chart of the SO schedule in Table 6.9. As seen from Figure 6.10, the SO schedule given in Table 6.9 takes 10 ms to complete an iteration. Thus, the throughput is $\frac{1}{10} = 100$ frames per second (fps). In Figure 6.10, the grey and white coloured boxes denote, if a processor is running at the frequency f_2 or f_1 respectively. Similarly, the dashed yellow coloured boxes refer to the frequency transition from f_1 to f_2, and vice versa. Please note that the processors π_2 and π_3 are in the same VFI, hence they always run at the same frequency. □

6.6 Translation of System Model to Hybrid Automata

Our framework consists of separate models of KiBaMs, a KiBaM scheduler, an SDF graph scheduler, and the processor application model. In this way, we divide the problem of evaluating the QoS in terms of power source, tasks and resources. In this section, we describe the translation of an SDF graph scheduler along with a processor application model and KiBaMs to HA using UPPAAL SMC.

Given an SDF graph $G = (A, D, \mathsf{Tok}_0, \tau)$ mapped on a processor application model $(\Pi, \zeta, F, I_{occ}, I_{idle}, I_{tr}, T_{tr}, \tau_{act})$ powered by a KiBaM system $\mathcal{KS} = (B, Cap)$, we generate a parallel composition of HA:

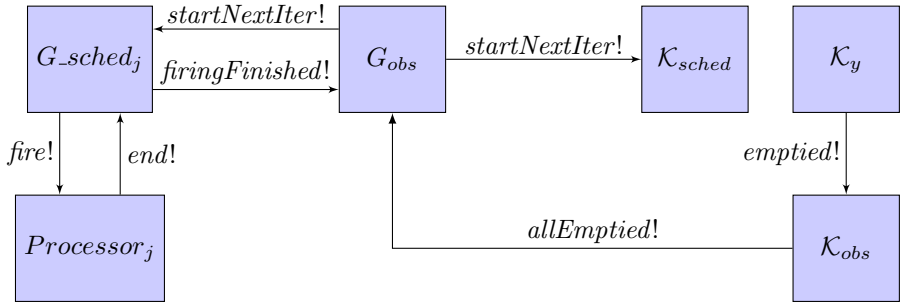


Figure 6.11: Interactions between HA of different components

$$\mathcal{K}_{sched} \| \mathcal{K}_1 \| \ldots, \| \mathcal{K}_m \| G\_sched_1 \| \ldots, \| G\_sched_n \| \\ Processor_1 \| \ldots, \| Processor_n \| G_{obs} \| \mathcal{K}_{obs}.$$

The brief explanation of each component is as follows.

- $\mathcal{K}_{sched}$: The automaton $\mathcal{K}_{sched}$ models the scheduling scheme of KiBaMs. We consider the round-robin scheduling scheme, i.e., after every iteration, the next available KiBaM is chosen to serve for the next iteration.
- $\mathcal{K}_y$: The HA $\mathcal{K}_1, \ldots, \mathcal{K}_m$ model the KiBaMs $B = \{bat_1, \ldots, bat_m\}$.
- G_sched_j: We define a SO schedule for each processor. The HA G_sched_1, $\ldots$, G_sched_n implement these SO schedules on the processors $\Pi = \{\pi_1, \ldots, \pi_n\}$.
- $Processor_j$: The HA $Processor_1, \ldots, Processor_n$ model the processors $\Pi = \{\pi_1, \ldots, \pi_n\}$.
- G_{obs}: The SDF graph observer automaton G_{obs} counts if each processor has fired all its mapped actors, according to its SO schedule. Hence, this automaton determines when an iteration is finished.
- $\mathcal{K}_{obs}$: The KiBaM observer automaton $\mathcal{K}_{obs}$ observes if any KiBaM gets emptied. This automaton also observes when all KiBaMs get emptied, symbolising end of the system lifetime.

Figure 6.11 shows the interactions between HA of different components. In the start, G_sched_j maps an actor to processor $Processor_j$ according to its SO schedule, using the action fire. When an actor finishes its firing on $Processor_j$, it is informed back to G_sched_j by $Processor_j$ using the action end so that G_sched_j can map the next actor on $Processor_j$. When all actors in a SO schedule of $Processor_j$ are fired, G_sched_j conveys this information to G_{obs} using the action firingFinished. As a result, G_{obs} increments the number of finished iterations by one. Afterwards, G_{obs} asks G_sched_j to start executing the SO again via the action startNextIter. Similarly, G_{obs} also synchronises with $\mathcal{K}_{sched}$ using the same action startNextIter, so that $\mathcal{K}_{sched}$ activates the next available battery.

When a battery $\mathcal{K}_y$ is emptied, it informs to $\mathcal{K}_{obs}$ via the action emptied. When all batteries get emptied, $\mathcal{K}_{obs}$ informs G_{obs} via the action allEmptied about the end of the system lifetime.

The detailed translation of all components to HA, is given in Appendix A on page 199.

6.7 Experimental Evaluation via MPEG-4 Decoder

We evaluate QoS factors via the example of the MPEG-4 decoder in Figure 6.6 on page 135. The experimental setup consists of an MPEG-4 decoder mapped on Samsung Exynos 4210 processors $\Pi = \{\pi_1, \ldots, \pi_n\}$. The processors $\Pi = \{\pi_1, \ldots, \pi_n\}$ are provided charge by Samsung batteries $B = \{bat_1, \ldots, bat_m\}$ used in Samsung Galaxy Fame smartphones. The capacity of all $bat \in B$ is, $Cap(bat) = 1300$ mAh. The processors $\Pi = \{\pi_1, \ldots, \pi_n\}$ are available with two frequency levels (MHz) $f_2 = 1400$ and $f_1 = 1032.7$. Table 6.8 on page 137 shows idle and operating load currents of both KiBaMs $B = \{bat_1, bat_2\}$ at both frequencies. The supposed transition overhead (ms) of all Exynos 4210 processors is, $T_{tr}(\pi, f_2, f_1) = T_{tr}(\pi, f_1, f_2) = 1$. The supposed time overhead is $I_{tr}(\pi,f_2,f_1) = I_{tr}(\pi,f_1,f_2) = 0.5$ mA for all $\pi \in \Pi$. We evaluate the completed number of video frames with respect to the QoS aspects of varying (1) frames per second (throughput); (2) number of processors; and (3) batteries. Similarly, for the same factors, we assess adequate battery capacities. Please see Figures 6.13-6.20 for results.

6.7.1 Varying Frames per Second

For 6 Exynos 4210 processors $\Pi = \{\pi_1, \ldots, \pi_6\}$ served by two batteries $B = \{bat_1, bat_2\}$, we consider different SO schedules, as given in Table 6.12. For varying frames per second (fps) constraint, Figure 6.13 shows the total number of video frames completed. At tighter performance constraints (i.e., higher fps), the idle time of processors is not sufficient to move to low power state. As a result, the batteries are drained more rapidly. Thus, we achieve less number of frames. Alternatively, if we require fewer fps from an MPEG-4 decoder, then the battery lifetime increases.

For the same SO schedules, Figure 6.14 shows the minimum initial required capacity $Cap(bat_1)$ for KiBaM $bat_1 \in B$ to complete 1000 video frames. It can be seen from Figure 6.14 that if we relax the fps constraint, the minimum required capacity also decreases.

Nevertheless, if the video quality is enhanced from 125 to 200 fps, then the increase in required initial battery capacity is relatively small equal to 84 mAh. However, the improvement in the video quality is considerable. Thus, higher performance can also be achieved at the expense of a small increase in battery capacities, leading to high-performance systems with less mass and volume. Hence, this method allows us to obtain a Pareto front by sweeping throughput constraints, for a fixed number of processors and batteries.

SO Schedule	Fps	π_1	π_2	π_3	π_4	π_5	π_6
S1	200	(FD-f_2)(VLD-f_2) (IDC-f_2)(RC-f_2)	(Idle-f_2)^2(VLD-f_2) (IDC-f_2)(Idle-f_2)	(Idle-f_2)^2(VLD-f_2) (IDC-f_2)(Idle-f_2)	(Idle-f_2)^2(VLD-f_2) (IDC-f_2)(Idle-f_2)	(Idle-f_2)^2(VLD-f_2) (IDC-f_2)(Idle-f_2)	(Idle-f_2)^3 (MC-f_2)(Idle-f_2)
S2	125	(FD-f_2)(VLD-f_2) (f_2-f_1)(IDC-f_1) (f_1-f_2)(RC-f_2)	(Idle-f_2)^2(VLD-f_2) (f_2-f_1)(IDC-f_1) (f_1-f_2)(Idle-f_2)	(Idle-f_2)^2(VLD-f_2) (f_2-f_1)(IDC-f_1) (f_1-f_2)(Idle-f_2)	(Idle-f_2)^2(VLD-f_2) (f_2-f_1)(IDC-f_1) (f_1-f_2)(Idle-f_2)	(Idle-f_2)^2(VLD-f_2) (f_2-f_1)(IDC-f_1) (f_1-f_2)(Idle-f_2)	(Idle-f_2)^3(f_2-f_1) (MC-f_1)(f_1-f_2) (Idle-f_2)
S3	111	(FD-f_2)(f_2-f_1) (VLD-f_1)(IDC-f_1) (f_1-f_2)(RC-f_2)	(Idle-f_2)^2(f_2-f_1) (VLD-f_1)(IDC-f_1) (f_1-f_2)(Idle-f_2)	(Idle-f_2)^2(f_2-f_1) (VLD-f_1)(IDC-f_1) (f_1-f_2)(Idle-f_2)	(Idle-f_2)^2(f_2-f_1) (VLD-f_1)(IDC-f_1) (f_1-f_2)(Idle-f_2)	(Idle-f_2)^2(f_2-f_1) (VLD-f_1)(IDC-f_1) (f_1-f_2)(Idle-f_2)	(Idle-f_2)^2(f_2-f_1) (Idle-f_1)^2(MC-f_1) (f_1-f_2)(Idle-f_2)
S4	100	(FD-f_2)(f_2-f_1) (VLD-f_1)(IDC-f_1) (RC-f_1)(f_1-f_2)	(Idle-f_2)^2(f_2-f_1) (VLD-f_1)(IDC-f_1) (Idle-f_1)^2(f_1-f_2)	(Idle-f_2)^2(f_2-f_1) (VLD-f_1)(IDC-f_1) (Idle-f_1)^2(f_1-f_2)	(Idle-f_2)^2(f_2-f_1) (VLD-f_1)(IDC-f_1) (Idle-f_1)^2(f_1-f_2)	(Idle-f_2)^2(f_2-f_1) (VLD-f_1)(IDC-f_1) (Idle-f_1)^2(f_1-f_2)	(Idle-f_2)^2(f_2-f_1) (Idle-f_1)^2(MC-f1) (Idle-f_1)^2(f_1-f_2)
S5	91	(f_2-f_1)(FD-f_1) (VLD-f_1)(IDC-f_1) (RC-f_1)(f_1-f_2)	(f_2-f_1)(Idle-f_1)^3 (VLD-f_1)(IDC-f_1) (Idle-f_1)^2(f_1-f_2)	(f_2-f_1)(Idle-f_1)^3 (VLD-f_1)(IDC-f_1) (Idle-f_1)^2(f_1-f_2)	(f_2-f_1)(Idle-f_1)^3 (VLD-f_1)(IDC-f_1) (Idle-f_1)^2(f_1-f_2)	(f_2-f_1)(Idle-f_1)^3 (VLD-f_1)(IDC-f_1) (Idle-f_1)^2(f_1-f_2)	(f_2-f_1)(Idle-f_1)^5 (MC-f_1)(Idle-f_1)^2 (f_1-f_2)

Table 6.12: SO schedules for varying number of video frames on 6 processors π_1, π_2, π_3, π_4, π_5, and π_6

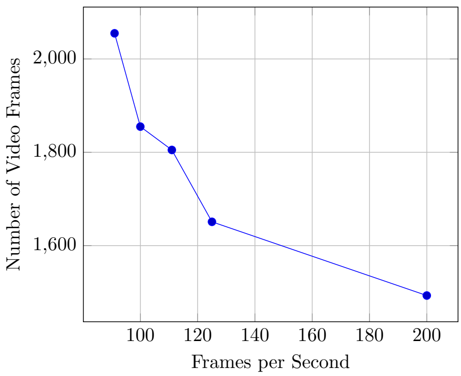


Figure 6.13: System lifetime against varying fps

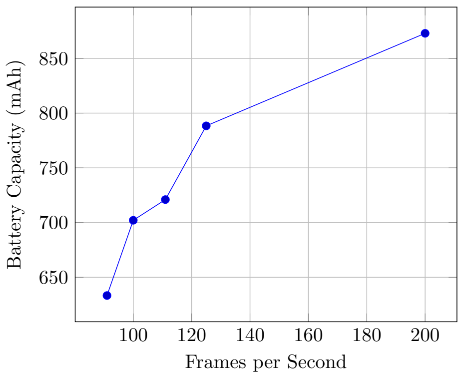


Figure 6.14: Minimum required capacity for bat_1

6.7.2 Varying Number of Processors

We consider different SO schedules given in Table 6.15, all yielding 71 fps. Figure 6.16 shows the total number of video frames completed for varying number of processors. As we can see from Figure 6.16, for the same batteries' capacities, higher number of processors achieve more or equal number of frames. The reason is that, if we reduce the number of processors, then the same amount of work is done on fewer processors to attain the same throughput, resulting in shorter idle times. Therefore, battery charge is consumed more rapidly, if the number of processors are reduced.

For the same SO schedules considered earlier in Table 6.15, Figure 6.17 shows the minimum required capacity $Cap(bat_1)$ for KiBaM $bat_1 \in B$ to complete 1000 video frames. The results reiterate the earlier conclusions in Figure 6.16 that, to

SO Schedule	π_1	π_2	π_3	π_4	π_5	π_6
S1	(FD-f_2)(VLD-f_2) (VLD-f_2)(VLD-f_2) (VLD-f_2)(VLD-f_2) (IDC-f_2)(IDC-f_2) (IDC-f_2)(IDC-f_2) (IDC-f_2)(MC-f_2) (RC-f_1)	-	-	-	-	-
S2	($f_2 \rightarrow f_1$)(FD-f_1) ($f_1 \rightarrow f_2$)(VLD-f_2) (VLD-f_2)(VLD-f_2) (IDC-f_2)(IDC-f_2) (MC-f_2)(RC-f_2) (MC-f_2)	($f_2 \rightarrow f_1$)(Idle-f_1)^3 ($f_1 \rightarrow f_2$)(VLD-f_2) (VLD-f_2)(IDC-f_2) (IDC-f_2)(IDC-f_2) (Idle-f_2)^3	-	-	-	-
S3	($f_2 \rightarrow f_1$)(FD-f_1) (VLD-f_1)($f_1 \rightarrow f_2$) (VLD-f_2)(IDC-f_2) (IDC-f_2)(RC-f_2) (Idle-f_2)^2	($f_2 \rightarrow f_1$)(Idle-f_1)^3 (VLD-f_1)($f_1 \rightarrow f_2$) (VLD-f_2)(IDC-f_2) (MC-f_2)(Idle-f_2)^3	($f_2 \rightarrow f_1$)(Idle-f_1)^3 (VLD-f_1)($f_1 \rightarrow f_2$) (IDC-f_2)(Idle-f_2)^4	-	-	-
S4	($f_2 \rightarrow f_1$)(FD-f_1) (VLD-f_1)(VLD-f_1) (IDC-f_1)($f_1 \rightarrow f_2$) (RC-f_2)(Idle-f_2)^3	($f_2 \rightarrow f_1$)(Idle-f_2)^3 (VLD-f_1)(IDC-f_1) (IDC-f_1)($f_1 \rightarrow f_2$) (Idle-f_2)^3	($f_2 \rightarrow f_1$)(Idle-f_2)^3 (VLD-f_1)(IDC-f_1) (MC-f_1)($f_1 \rightarrow f_2$) (Idle-f_2)^3	($f_2 \rightarrow f_1$)(Idle-f_2)^3 (VLD-f_1)(IDC-f_1) (Idle-f_2)^2($f_1 \rightarrow f_2$) (Idle-f_2)^3	-	-
S5	($f_2 \rightarrow f_1$)(FD-f_1) (VLD-f_1)(IDC-f_1) (MC-f_1)(RC-f_1) ($f_1 \rightarrow f_2$)(Idle-f_2)^2	($f_2 \rightarrow f_1$)(Idle-f_1)^3 (VLD-f_1)(IDC-f_1) (Idle-f_1)^3($f_1 \rightarrow f_2$) (Idle-f_2)^2	($f_2 \rightarrow f_1$)(Idle-f_1)^3 (VLD-f_1)(IDC-f_1) (Idle-f_1)^3($f_1 \rightarrow f_2$) (Idle-f_2)^2	($f_2 \rightarrow f_1$)(Idle-f_1)^3 (VLD-f_1)(IDC-f_1) (Idle-f_1)^3($f_1 \rightarrow f_2$) (Idle-f_2)^2	($f_2 \rightarrow f_1$)(Idle-f_1)^3 (VLD-f_1)(IDC-f_1) (Idle-f_1)^3($f_1 \rightarrow f_2$) (Idle-f_2)^2	-
S6	($f_2 \rightarrow f_1$)(FD-f_1) (VLD-f_1)(IDC-f_1) (RC-f_1)(Idle-f_1)3 ($f_1 \rightarrow f_2$)	($f_2 \rightarrow f_1$)(Idle-f_1)^3 (VLD-f_1)(IDC-f_1) (Idle-f_1)^5($f_1 \rightarrow f_2$)	($f_2 \rightarrow f_1$)(Idle-f_1)^3 (VLD-f_1)(IDC-f_1) (Idle-f_1)^5($f_1 \rightarrow f_2$)	($f_2 \rightarrow f_1$)(Idle-f_1)^3 (VLD-f_1)(IDC-f_1) (Idle-f_1)^5($f_1 \rightarrow f_2$)	($f_2 \rightarrow f_1$)(Idle-f_1)^3 (VLD-f_1)(IDC-f_1) (Idle-f_1)^5($f_1 \rightarrow f_2$)	($f_2 \rightarrow f_1$)(Idle-f_1)^3 (IDC-f_1)(Idle-f_1)^7 ($f_1 \rightarrow f_2$)

Table 6.15: SO schedules for varying number of processors. All of these schedules yield 71 fps.

achieve the same throughput, fewer processors carrying out the work of same magnitude require larger battery capacities.

Hence, using this method, a system designer can estimate QoS for different design alternatives. For instance, in our running example, one can clearly see that we can achieve same throughput for 4 processors, as 5, without requiring extra capacities for batteries. Therefore, we may not need more processors in our platform, and reach a certain throughput with fewer number of processors, and same batteries' capacities, contributing to low-cost embedded systems with reduced mass and volume.

6.7.3 Varying Number of Batteries

Let us consider that we have 6 Exynos 4120 processors $\Pi = \{\pi_1, \ldots, \pi_6\}$. We consider a SO schedule producing 71 fps, as given in Table 6.18. For varying batteries, Figure 6.19 and 6.20 shows the total number of video frames completed, and the minimum required capacity $Cap(bat_1)$ for battery $bat_1 \in B$ to complete 1000 video frames respectively. As it can be seen from Figure 6.19, increasing the number of batteries improves the attainable number of video frames linearly.

However, if we analyse Figure 6.20, we can see that increasing the number of batteries does not reduce the minimum required battery capacities at a linear rate. Therefore, we can conclude that, having fewer batteries with larger capacities is more beneficial than higher number of batteries with smaller capacities. This achieves the low-cost and high-performance systems.

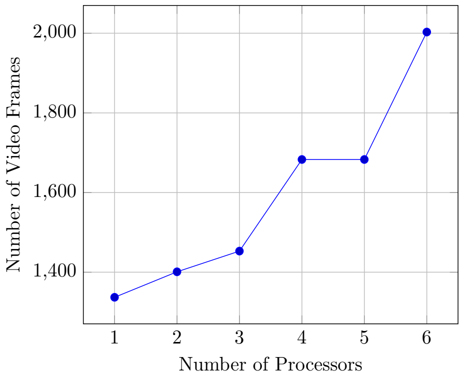


Figure 6.16: System lifetime against varying number of processors

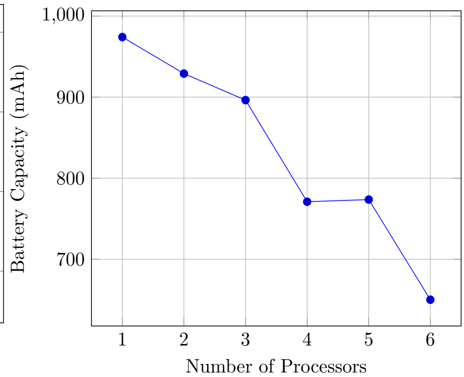


Figure 6.17: Minimum required capacity for bat_1

π_1	π_2	π_3	π_4	π_5	π_6
$(f_2 \rightarrow f_1)$(FD-f_1)(VLD-f_1)(IDC-f_1)(RC-f_1)(Idle-f_1)3 $(f_1 \rightarrow f_2)$	$(f_2 \rightarrow f_1)$(Idle-f_1)^3 (VLD-f_1)(IDC-f_1)(Idle-f_1)^5$(f_1 \rightarrow f_2)$	$(f_2 \rightarrow f_1)$(Idle-f_1)^3 (VLD-f_1)(IDC-f_1)(Idle-f_1)^5$(f_1 \rightarrow f_2)$	$(f_2 \rightarrow f_1)$(Idle-f_1)^3 (VLD-f_1)(IDC-f_1)(Idle-f_1)^5$(f_1 \rightarrow f_2)$	$(f_2 \rightarrow f_1)$(Idle-f_1)^3 (VLD-f_1)(IDC-f_1)(Idle-f_1)^5$(f_1 \rightarrow f_2)$	$(f_2 \rightarrow f_1)$(Idle-f_1)^3 (IDC-f_1)(Idle-f_1)^7 $(f_1 \rightarrow f_2)$

Table 6.18: SO schedule for varying number of batteries yielding 71 fps

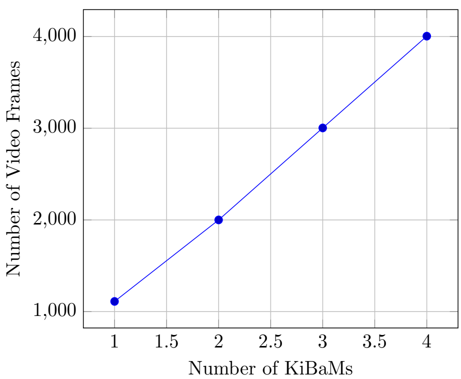


Figure 6.19: System lifetime against varying number of batteries

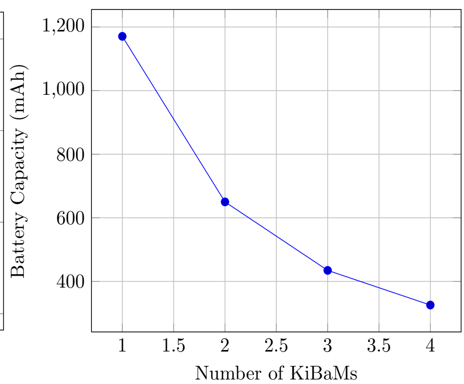


Figure 6.20: Minimum required capacity for bat_1

6.7.4 Comparison with PTA-KiBaM

In this subsection, we compare our approach (HA-KiBaMs) with a PTA-based approach (PTA-KiBaM) [JHBK09]. In PTA-KiBaM, the behaviour of batteries is based on a discretised version of the KiBaM, and is modelled as priced timed automata (PTA). For a given load, the model checker UPPAAL CORA is utilised to search the whole state-space and to generate optimal battery schedules. However, this approach suffers serious scalability issues. As increasing the initial batteries'

SO Schedule	π_1	π_2	π_3	π_4	π_5	π_6
S1	($f_2 \rightarrow f_1$)(FD-f_1) (VLD-f_1)($f_1 \rightarrow f_2$) (VLD-f_2)^2(IDC-f_2) (IDC-f_2)^2(RC-f_2)	($f_2 \rightarrow f_1$)(Idle-f_1)^3 (VLD-f_1)($f_1 \rightarrow f_2$) (VLD-f_2)(IDC-f_2) (MC-f_2)(Idle-f_2)	($f_2 \rightarrow f_1$)(Idle-f_1)^3 (VLD-f_1)($f_1 \rightarrow f_2$) (IDC-f_2)(IDC-f_2) (Idle-f_2)2	-	-	-
S2	(f_2-f_1)(FD-f_1) (VLD-f_1)(f_1-f_2) (IDC-f_2)^2(RC-f_2)	(Idle-f_2)^3(f_2-f_1) (VLD-f_1)(MC-f_1) (f_1-f_2)(Idle-f_2)	(Idle-f_2)^3(f_2-f_1) (VLD-f_1)(MC-f_1) (f_1-f_2)(Idle-f_2)	(Idle-f_2)^3(VLD-f_2)^2 (IDC-f_2)^2(Idle-f_2)^3	-	-
S3	(f_2-f_1)(FD-f_1) (VLD-f_1)(IDC-f_1) (RC-f_1)(f_1-f_2)	(f_2-f_1)(Idle-f_1)^3 (VLD-f_1)(IDC-f_1) (Idle-f_1)^2(f_1-f_2)	(f_2-f_1)(Idle-f_1)^3 (VLD-f_1)(IDC-f_1) (Idle-f_1)^2(f_1-f_2)	(f_2-f_1)(Idle-f_1)^3 (VLD-f_1)(IDC-f_1) (Idle-f_1)^2(f_1-f_2)	(f_2-f_1)(Idle-f_1)^3 (VLD-f_1)(IDC-f_1) (Idle-f_1)^2(f_1-f_2)	(f_2-f_1)(Idle-f_1)^5 (MC-f_1)(Idle-f_1)^2 (f_1-f_2)

Table 6.21: SO schedules for comparison of our approach (HA-KiBaMs) with the PTA-based approach (PTA-KiBaM)

capacities leads to searching the bigger state-space, this approach only allows to model limited batteries' capacities. Furthermore, this approach requires to discretise the temporal dimension, which limits the accuracy of this approach. In contrast, we use hybrid automata to model the continuous behaviour of batteries. This leads us to analyse the behaviour of KiBaMs without discretising time. Furthermore, following this approach, we can make use of highly scalable Monte Carlo simulations, over hybrid automata. It is worth mentioning that the PTA-KiBaM [JHBK09] analyses the completed number of tasks, instead of iterations. However, as iterations are the key metric in SDF graphs, we also compare both techniques in terms of the completed number of iterations.

Let us consider the example of an MPEG-4 decoder in Figure 6.6 on page 135. We assume that we have two batteries, i.e., $B = \{bat_1, bat_2\}$. Table 6.22 shows the completed number of video frames for the arbitrary SO schedules in Table 6.21, calculated using both methods. The experiments were run on a dual-core 2.8 GHz machine with 8 GB RAM.

Columns 3-8 in Table 6.22 show the completed number of iterations and computation time (ms), calculated using both methods, against different battery capacities (mAh) in Columns 1-2. The results in Table 6.22 show that HA-KiBaM achieves the same results as PTA-KiBaM except S2. The reason of not producing the same results in S2 is that PTA-KiBaM allows to change the active battery during the iteration. Whereas, we consider a specific scheduling scheme, where we change the battery after an iteration is finished.

However, the biggest advantage of HA-KiBaM is the scale of capacities it can handle. As Table 6.22 shows, PTA-KiBaM can only handle very small battery capacities that are able to finish not more than one video frame. This makes PTA-KiBaM impracticable for modern-day systems, as opposed to our method that scales to much larger capacities (see Section 6.7). Furthermore, PTA-KiBaM requires considerably longer computation time than HA-KiBaM. Please note that zero in Table 6.22 means that the battery capacities are not enough, even to finish one iteration.

In addition to the battery capacities, our method also scales better to the number of batteries. Table 6.23 compares the iterations completed for varying

$Cap(bat_1)$	$Cap(bat_2)$	S1 (computation time)		S2 (computation time)		S3 (computation time)	
		PTA-KiBaM	HA-KiBaM	PTA-KiBaM	HA-KiBaM	PTA-KiBaM	HA-KiBaM
1.25×10^{-4}	1.25×10^{-4}	0 (520)	0 (28)	0 (200)	0 (46)	0 (2130)	0 (51.4)
2.5×10^{-4}	2.5×10^{-4}	0 (510)	0 (55)	1 (41060)	0 (48)	*Out of Memory*	0 (52.7)
3.75×10^{-4}	3.75×10^{-4}	*Out of Memory*	2 (62)	1 (14810)	0 (49)	*Out of Memory*	2 (52.8)
5×10^{-4}	5×10^{-4}	*Out of Memory*	4 (64)	*Out of Memory*	2 (49)	*Out of Memory*	4 (54.1)

Table 6.22: Comparison of two approaches with respect to varying battery capacities. Our approach performs better in terms of capacities and computation time.

Number of Batteries	HA-KiBaM	PTA-KiBaM
1	1	N/A
2	4	*Out of Memory*
3	6	*Out of Memory*
4	12	*Out of Memory*
5	15	*Out of Memory*
6	18	*Out of Memory*
7	21	*Out of Memory*
8	24	*Out of Memory*
9	27	*Out of Memory*
10	30	*Out of Memory*

Table 6.23: Comparison of two approaches with respect to number of batteries. Our approach scales better in terms of number of batteries.

number of batteries for both methods. For this experiment, we consider SO schedule S3, and $Cap(bat) = 5\times10^{-4}$ mAh for all $bat \in B$.

6.8 Model Checking via MPEG-4 Decoder

In this section, we demonstrate the analysis of functional properties, using the UPPAAL SMC model checker and its query language. We consider the case study of an MPEG-4 decoder in Figure 6.6 on page 135 mapped on Exynos 4210 processors, and powered by a KiBaM system.

Parallel firings of actors

We can check whether any actors can fire in parallel within time T. For example, actors p and q mapped on the processors π_1 and π_2 respectively, can

fire in parallel if the following query **Pr**[<= T](<> activeActor_1 == p and activeActor_2 == q) evaluates to true. Here, activeActor_j represents the actor currently mapped on the processor $\pi_j \in \Pi$. In our experiment, $p = MC$ and $q = RC$. As these two actors cannot fire in parallel, UPPAAL SMC answers that the probability for this query to hold is [0,0.0973938] with 0.95 confidence, which means that this property is not satisfied.

Same running frequency in an VFI

We can also check safety properties such as, at a given time, all processors belonging to the same VFI should not run at the different frequency within time T. For this purpose, we create a variable named curr_freq available to all processors, that keeps account of current running frequency of each processor. If we have two processors π_1 and π_2 in a same VFI, then we check the query **Pr**[<= T]([] Processor_1.curr_freq == Processor_2.curr_freq) to verify this property. UPPAAL SMC answers that the probability for this query to hold is [0.902606,1] with 0.95 confidence, which means that this query is satisfied.

Fair Battery Scheduling

We can also verify if a new battery is selected after each iteration. Let us assume that we have two batteries, and an integer empty_count to count the number of empty batteries. Let us further assume that we have a variable bound_bj to keep account of bound charge of a KiBaM $bat_j \in B$. We run the query **Pr**[<= T]([] bound_b2 - bound_b1 < n and empty_count < 2) that determines if within time T, the difference between the bound charge of two batteries does not exceed more than a certain amount, and each battery gets a fair chance to recover its bound charge. The probability for this property to hold is [0.902606,1] with 0.95 confidence. In our experiment, n is 4.

Active Number of Batteries

Similarly, we can also check that no more than one battery should be active within any given time T. Let us assume that we have two batteries, and boolean variables b1_active and b2_active is assigned to each battery respectively, to check if that battery is active. To verify this property, we use the query **Pr**[<= T](<> b1_active == true and b2_active == true). The probability for this property to hold is [0,0.0973938] with 0.95 confidence, which means that this property is not satisfied.

6.9 Conclusions

With the growing gap between the energy demand and battery densities, compact methods for guaranteeing QoS of multiple KiBaMs are still needed. We have presented a novel technique to predict system lifetime for SDF-modelled streaming applications, mapped on processors equipped with energy reduction techniques and powered by multiple batteries. This provides us with a best trade-off between the throughput, the number of processors and batteries. The

batteries are modelled as a hybrid system, which has the advantage of being accurate.

Part III

Modelling and Validation

CHAPTER 7

Model-Driven Engineering for Dataflow Applications

Abstract

The previous chapters have presented techniques of analysis and scheduling dataflow applications mapped on hardware platforms. This practice of separating software and hardware is known as HW-SW co-design. We have seen in earlier chapters that the models describe the behaviour of a system in an accurate and unambiguous way. Furthermore, modelling at a right abstract level, helps to understand complex systems such as concurrent ones, and their solutions.

What we have not presented in chapters 4 - 6 is an explanation of the *modelling approach* used for HW-SW co-designing of dataflow applications and platform application models. Furthermore, it is yet to be described how models in the model-checking domain are obtained. One may argue that the modelling approach is irrelevant as long as the models are correct. However, in today's agile world, we are witnessing that requirements change rapidly, there is always a competition to introduce new products/functionalities, and shortened time-to-market is required. Therefore, an automated modelling approach is needed which satisfies modularity, extensibility, and interoperability requirements. Model-Driven Engineering (MDE) is a prominent paradigm that, by treating models and model transformations as first-class citizens, helps to fulfil these requirements.

In this chapter, we present a state-of-the-art MDE-based framework for HW-SW co-design of dataflow applications. In the framework, we introduce a reusable set of three coherent metamodels for creating models concerning SDF graphs, platform application models (PAMs) and allocation of SDF tasks to PAMs. The framework also contains model transformations that cast these models into model-checking domain. We demonstrate how our framework satisfies the requirements of modularity, extensibility, and interoperability with a case study.

About this chapter: The current chapter is based on the paper "A Model-Driven Framework for Hardware-Software Co-design of Dataflow Applications", which was published at CyPhy 2016 [AYRS16a]. An extended report on the work was published at University of Twente Eprints [AYRS16b]. The original paper largely remains the same.

7.1 Introduction

HW-SW Co-design. Hardware-software (HW-SW) co-design is a successful engineering practice in the development of embedded and cyber-physical systems by the manufacturing companies of these systems [DS96]. Key insight is that HW and SW should be designed and evaluated together. In this way, the synergy of hardware and software is exploited to meet system-level objectives. For instance, HW-SW co-design allows exploring design alternatives and finding more efficient solutions, and helps to improve the development cost and time-to-market. Nowadays, HW-SW co-design has become common practice in the domain of automation [BGJ+02], avionics [CCE+99], industrial design [Mic96], automotive [KKN96], home appliances [SKH+97], mobile devices [vvK05], and many more [Tei12]. To support HW-SW co-design and its validation and verification, a large number of tools has been developed, e.g., Octopus [BHRV13], Ptolemy [BHLM94], COMPLEX [GHH+13], VeriSTA [HG13], SystemCoDesigner [KSS+09], Real-time Calculus [TCN00], and CoWare [RVBM96].

Challenges in HW-SW Co-design. As mentioned in the recent paper [Tei12], the challenges in HW-SW co-design include increasing complexity of modern-day applications due to concurrency, energy-constraints, heterogeneous multiprocessor architectures, etc. To deal with these challenges, it has been recognised that a HW-SW co-design approach must have the following features [BHRV13, GHH+13, HG13, BHLM94].

- *Modularity* [BHRV13, GHH+13]: The hardware platform, software application, and mapping of this application to the platform are not modularly and explicitly expressed as separate models. Hence, the final analysis model is a combined representation of these three aspects. As a result, any change occurring in any of these points will require modifications in the combined representation, even though the three points are largely independent from each other. Furthermore, if one wishes to analyse a particular software application on various platforms or a different mapping to the same platform, there is no hardware model that is explicitly defined as a module to be reused. A similar problem occurs if one wishes to test several applications on a particular platform. This suggests that, for the sake of modularity, the model should separate the aspects of hardware, software and their mappings: modules targeting different concerns are better maintainable and reusable.

- *Interoperability*[BHRV13, BHLM94, Tei12]: HW-SW co-designing often involves design groups from different disciplines, including firmware, operating system (OS), and application developers on the software side, as well as hardware developers and chip designers. These groups use different tools, with different models of computation and based on different programming languages. Systematic support for interoperability is required for flexible orchestration between such tools to work together.

- *Extensibility* [GHH+13]: Evolution in the domain and accordingly changes in the requirements and design aspects are inevitable. The introduction of new hardware technologies and the adaptation of new design analysis techniques are two examples of such changes. The HW-SW co-designing approach should have convenient extension mechanisms allowing rapid implementation of possible future requirements sourcing from these changes.

Model-Driven Engineering. Existing approaches for HW-SW co-design do not fully satisfy all of these features. Model-Driven Engineering (MDE) is an approach that helps to fulfil all of the aforementioned requirements [VSB+13]. In MDE, the important concepts of the target domain are formally captured in a so-called *metamodel*. Separate metamodels for the domains of interest help to keep the design modular. All models are instances of a metamodel, or possibly an integrated set of metamodels. These models can be transformed to other models via *model transformations* defined at the metamodel level, which provides interoperability. Moreover, metamodels and model transformations can be systematically extended to satisfy future requirements.

Proposed Approach. This chapter presents a novel HW-SW co-design framework based on the principles of MDE. Our framework allows model-driven HW-SW co-design of SDF applications mapped on multiprocessor hardware platforms. To exemplify, we consider the approach in Chapter 5 where we utilised priced timed automata (PTA) for modelling SDF graphs and platform applications models (PAMs), and performing energy-optimal scheduling. In this chapter, we explain how SDF graph models and PAMs are obtained and transformed to PTA models in UPPAAL CORA format explained in Chapter 5, using MDE.

Our framework consists of the following three metamodels, as shown in Figure 7.1:

1. a metamodel for SDF graphs;
2. a metamodel for Platform Application Models (PAMs), which describe the processor types and their power levels, and the cost of switching between the power levels; and
3. a metamodel for expressing potential allocations of the tasks in an SDF graph to the processor types in a PAM.

As mentioned earlier, our framework considers the model checker UPPAAL CORA for generating energy-optimal schedules. Therefore, for supporting the generation of UPPAAL CORA models, we use an existing UPPAAL metamodel developed at the University of Paderborn [PAD]. Triples of models conforming to the three metamodels discussed above are transformed to UPPAAL CORA models automatically via *model transformations* in the framework.

Chapter 5 already described the principles of our method of using PTA for the purpose of energy optimisation. Rather the novelty of this chapter is the design prospect of using MDE. In Section 7.5, we provide a case study as an evidence to show how our framework satisfies the modularity, extensibility and interoperability requirements.

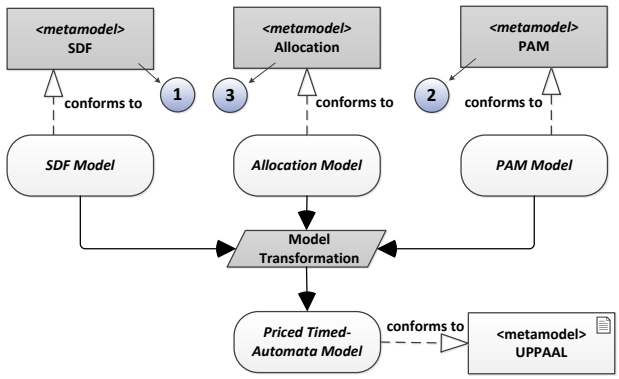


Figure 7.1: Overview of our framework. The framework contains metamodels of (1) SDF graphs, (2) PAMs, and (3) allocations. After models conforming to each metamodels are generated, they are automatically transformed to PTA models conforming to UPPAAL metamodel. The numbered metamodels are explained in the main text.

Chapter Contributions. The main contributions of this chapter are as follows:

- A reusable set of three coherent, extensible metamodels for HW-SW co-design centred the SDF formalism is developed[1].
- A set of model transformation rules is defined and applied. Using these rules, an automated tool which transforms from the dataflow domain to the PTA domain is developed, to compute energy-optimal schedules for dataflow applications.
- We have demonstrated that our fully automated framework provides modularity, extensibility and interoperability between tools, using a diverse set of scenarios for a case study. In particular:
  - *Modularity* is demonstrated by analysing an SDF graph on multiple platforms. We develop a new PAM for each platform, but we generate the SDF graph model only once, and reused it every time we need to analyse it on a different platform.
  - *Interoperability* is demonstrated by implementing model transformations between two different tools, i.e., SDF^3 (discussed below) and UPPAAL CORA. In this way, SDF^3 models can be automatically transformed to UPPAAL CORA models.

[1]All metamodels, model transformations, and case studies discussed in this chapter can be found at `https://github.com/utwente-fmt/COMET`. An instruction manual for replicating the experiments is also given in this repository.

  - *Extensibility* is demonstrated by extending PAMs with a *memory* element. This new extension takes memory read and write times of data required by the tasks into consideration.

- The complete set of the PTA models of the case study is provided.

Origins of the Chapter. This chapter was written in collaboration with Bugra M. Yildiz from University of Twente. The author defined and applied model transformations, and performed all case studies along with their analysis, while Bugra M. Yildiz developed metamodels and a visual editor for modelling PAMs.

Chapter Outline. The rest of the chapter is structured as follows: Section 7.2 discusses related work. Section 7.3 gives an overview of our framework and Section 7.4 describes the framework components in detail. Section 7.5 evaluates our framework using a case study, and Section 7.6 concludes the chapter.

7.2 Related Work

There exists a plethora of commercial and academic tools for HW-SW co-designing [BHRV13, GHH+13, HG13, BHLM94, TCN00, RVBM96, KSS+09]. However, most of these tools provide their own modelling and analysis mechanisms which makes it difficult to integrate them. Furthermore, most of these tools offer much less support for systematic extensions. Our framework improves upon these existing approaches by making use of model-driven engineering techniques with the advantages discussed above. In the following, we discuss different tools based on (1) HW-SW co-design and (2) MDE.

7.2.1 HW-SW Co-Design

In the study by Basten et al. [BHRV13], the Octopus Design-Space Exploration toolset is introduced to support model-driven design-space exploration for embedded systems. The toolset aims to support reuse and combined use of models between different domains and tools. The toolset contains a set of Java libraries for reading the design models written in their own textual specification language and then transforming them to other tools such as coloured Petri-Nets tools, SDF^3, and UPPAAL for design-space exploration. If any further transformations need to be defined for other tools, one has to implement these transformations using the Java libraries. Rather than Java, which is a general-purpose language, we use ETL, a domain-specific language for model transformation. Furthermore, the Octopus toolset does not provide any metamodels. The lack of metamodels and failure to use a model transformation language cause challenges in interoperability and maintainability of the toolset, which are in fact stated as a future directions of research in their study.

Ptolemy [BHLM94] is another well-known tool supporting experimentation with HW-SW co-design. Ptolemy allows detailed task-oriented design and simulation using various modelling techniques including SDF models, process

networks, continuous-time models, etc.. The models are designed using object-orientation approach in C++. Similar to Octopus, Ptolemy faces challenges in interoperability and maintainability due to the lack of metamodels and model transformations.

The work in [FGFR15, Fak16] aims to analyse timing bounds of SDF graphs, using the UPPAAL model checker. The method, similar to ours, extracts SDF models from SDF^3 and models hardware separately, and then generates a timed-automata model from them using a graphical user interface implemented using EMF. However, the method does not make use of any explicit metamodels or model transformations. As a result, it is not clear how future requirements such as integration of different tools and analysis approaches can be implemented.

Grüttner et al. proposes a reference framework to generate virtual executable prototypes from HW-SW co-designs and study different hardware platforms and power management strategy at early states of development [GHH+13]. The framework generates the prototypes in terms of executable models derived from formal UML/MARTE specifications. Grüttner et. al.'s framework follows the modularity principle by separating various concerns of the system through viewpoints while we achieve this through the use of metamodels for various concerns. In their paper, there is no explanation about if the reference framework offers any systematic extension mechanisms for future requirements.

Herber and Glesner proposed a framework for automatic verification of HW/SW co-designs using timed-automata models in [HG13]. The framework translates the HW-SW co-design implemented in SystemC to the timed-automata format of the UPPAAL model checker. This translation is automatically achieved by the SystemC Timed Automata Transformation Engine (STATE) that is specifically designed for SystemC-to-UPPAAL transformations. Since STATE is implemented in a general-purpose programming language, i.e., Java, and does not use model-driven engineering techniques, this limits to what extent the requirements of modularity, interoperability and extensibility are satisfied [VSB+13].

7.2.2 Model-Driven Engineering

The closest work to ours is presented in [BDD05] and [GMA+11]. In [BDD05], Bonde et al. apply MDE techniques to achieve HW-SW co-design of embedded systems for simulation and code generation purposes. They have developed a new model transformation engine and a declarative XML-based model transformation language for this engine. Using the transformation engine and the specified transformation definitions, they generate allocation models and subsequently code templates for simulation. Our framework differs from the work by Bonde et al. mainly in two points. Firstly, we are aiming to achieve analysis of the final models via model checking rather than simulation. Secondly, our choice of ETL as the model transformation language provides us better and more flexible functionality while defining and extending model transformations for various purposes, as explained in Section 7.3.3.

The MADES approach introduced in [GMA+11] supports validation, simulation and code generation of embedded systems using MDE techniques. They

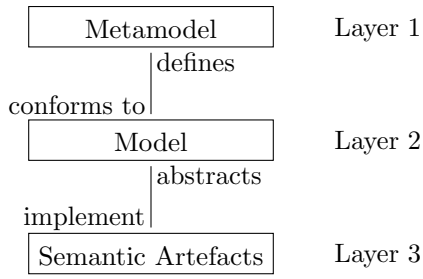


Figure 7.2: Metamodelling layered approach [Obj15]. Semantic artefacts at Layer 3 are implemented as models as Layer 2, which conform to the metamodels in Layer 1.

use the MADES modelling language to create design and analysis models. Our work is in a different direction since it is focused more on dataflow applications and heterogeneous processors.

7.3 The Model-Driven Framework

7.3.1 Model-Driven Engineering

Models are powerful artefacts to express behaviour, structure and other properties in many domains such as mathematics, engineering, and other natural sciences. Model-Driven Engineering (MDE) is a software engineering paradigm that considers models not only as documentation, but also adopts them as the basic abstraction to be used throughout all engineering disciplines and in any application domain [dS15]. The models in MDE are closer to some particular domain concepts rather than the computing concepts. These models are considered equivalent to code, since they are formally defined and have execution semantics.

To define models, we need to specify their language as a model of these models at a more abstract level, so-called *metamodel*s. In other words, metamodelling is the modelling of models. In their common use, metamodels define the permitted structure, to which models must adhere. Therefore, metamodels describe the syntax of models [SRVK10]. The layered approach used in metamodelling [Obj15] is shown in Figure 7.2. In Figure 7.2, the artefact in each layer conforms to, or is abstracted by the adjacent layer. Therefore, semantic artefacts in Layer 3 are abstracted in models defined in Layer 2, which further conform to the metamodels given in Layer 1. For example, some information —such as a task graph— is a semantic artefact, located at Layer 3. The XML file representing this information is a model of this semantic artefact, which is located at Layer 2. Finally, the DTD or XSD schema that this XML file conforms to can be counted as a metamodel, located at Layer 1. The design process that utilises metamodelling first abstracts the concepts of some domain or application (Layer 3) into the metamodel (Layer 1). Afterwards, the models (Layer 2) defining the domain conforming to their metamodels are generated.

MDE helps us to satisfy modularity, interoperability, and extensibility requirements in the following way:

- There can be multiple domains in a project. In MDE, the concepts and their relationships in each of these domains can be captured separately into different metamodels. In this way, various concerns of these domains can be kept *modular*. Models that are instances of these metamodel become better reusable.

- The tools used in a project have typically their own concrete syntax for their models, such as XML, JSON or plain text. MDE allows *interoperability* between different domains (and tools in these domains) via *model transformation*s. Model transformations provide interoperability and furthermore save effort and reduce errors by automating the model derivation and modification process.

- Technologies used for applying MDE provide the following mechanisms for systematic *extensibility*: Extending existing metamodels and model transformations, introduction of new metamodels to existing tool chains.

7.3.2 Overview of the Model-Driven Framework

Figure 7.3 gives a detailed overview of our framework introduced earlier in Figure 7.1. The HW-SW co-design of the application consists of the first four steps:

- In step 1, an SDF model of the software application is created using the SDF^3 tool in an XML format specific to the tool.

- In step 2, the SDF model is automatically transformed to an SDF model that conforms to the metamodel we defined for SDF graphs.

- In step 3, a hardware platform model is created using *PAM Visual Editor* that is a graphical editor for specifying Platform Application Models (PAMs). This model conforms to the PAM metamodel.

- In step 4, an allocation model is created for specifying the mapping of the tasks in the SDF model to the processor types in the PAM.

The analysis of the co-design for energy-optimal schedules is conducted using the UPPAAL CORA model checker. This is achieved in the last three steps:

- In step 5, the co-design is transformed to a PTA model that conforms to the UPPAAL metamodel.

- In step 6, the PTA model is transformed to the format accepted by the model checker.

- In step 7, we compute the energy-optimal schedule using the UPPAAL CORA model checker.

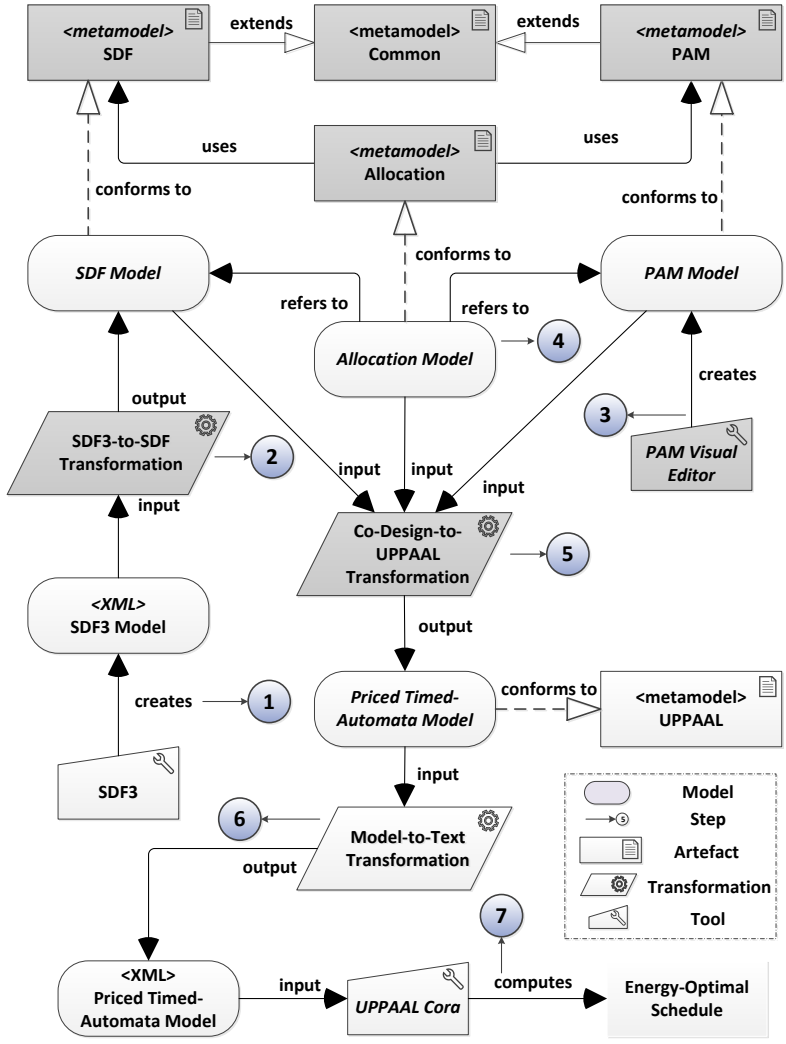


Figure 7.3: Detailed overview of our framework. The elements with dark background colour represent the new contributions in this thesis. The numbered steps are explained in the main text.

Here, the steps 1, 3, and 4 are manual, and the steps 2, 5, 6, and 7 are automatic.

Further details related to the elements of the framework are described in Section 7.4.

Although the steps in Figure 7.3 show a general guideline for a HW-SW co-design of a system from scratch, a different strategy can be adopted according to the requirements of the system design. For example, if a system designer needs to analyse how a software application runs on various hardware platforms, s/he can create an SDF model by follow steps 1 and 2 only once and then create

several PAM models by conducting step 3 multiple times.

7.3.3 Tooling Choices

To realise the model-driven approach, we have created metamodels using ECore in the Eclipse Modelling Framework (EMF) [SBMP08]. EMF provides a plethora of plugins to support various functions, such as querying, validation, and transformation of EMF models. For instance, using the EuGENia plugin [KRPP09], we have created the PAM Visual Editor based on Eclipse Graphical Editing Framework (GMF). To ensure the well-formedness of the metamodels, we have defined Object Constraint Language (OCL) rules. OCL is a precise text language to express constraints on metamodels that cannot otherwise be expressed by diagrammatic notation [Obj12].

The model transformations have been implemented using Epsilon Transformation Language (ETL) [KPP08], which is one of the domain-specific languages provided by the Epsilon framework. ETL is a hybrid model transformation language that combines declarative mapping style with imperative features to support definition of complex transformations. ETL supports many input-to-many output model transformations; it also allows the users to inherit, import and reuse other Epsilon modules in the transformations.

7.4 Details of the Model-Driven Framework

This section presents our concrete instantiation of the model-driven framework by describing our modelling choices in some detail. We recall the formal (mathematical) definitions of the domain concepts from Chapter 5 and discuss how we have chosen to translate them to metamodel elements.

7.4.1 SDF Graphs

Definition and Metamodel

Definition 7.1. *An* SDF graph *is a tuple* $G = (A, D, \mathsf{Tok}_0)$ *where*

- A *is a finite set of actors,*
- $D \subseteq A^2 \times \mathbb{N}^2$ *is a finite set of channels, and*
- $Tok_0 : D \to \mathbb{N}$ *denotes the initial number of tokens on each channel.*

Some notation: given an SDF graph G as above, the sets of *input* and *output channels* of an actor $a \in A$ are defined respectively as $In(a) = \{(b, a, p, q) \in D \mid b \in A, p, q \in \mathbb{N}\}$ and $Out(a) = \{(a, b, p, q) \in D \mid b \in A, p, q \in \mathbb{N}\}$.

The state of an SDF graph is represented by a function from D to $\mathbb{N}$ called a *token distribution*. For instance, the function Tok_0 in Definition 7.1 is such a distribution — namely, the initial one. Actor a can fire if each input channel $(b, a, p, q) \in In(a)$ contains at least q tokens in the current distribution; firing it removes those tokens, and ends by producing p tokens on each output channel $(a, b, p, q) \in Out(a)$. We consider SDF graphs to be untimed.

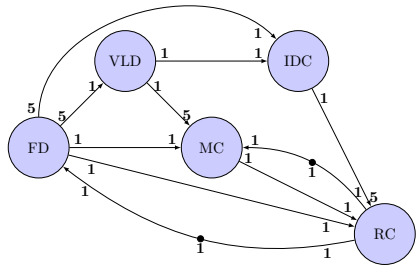


Figure 7.4: SDF graph of an MPEG-4 decoder

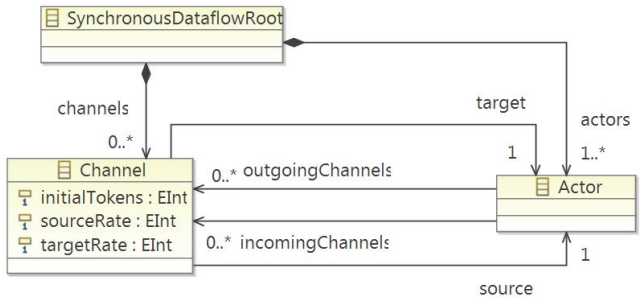


Figure 7.5: SDF Metamodel. *SynchronousDataflowRoot* is the root of a model, *Actor* corresponds to the set of actors, and *Channel* corresponds to the set of channels.

Example 7.2. Figure 7.4 shows the SDF graph of an MPEG-4 decoder. The actors A={FD, VLD, IDC,RC, MC} represent individual tasks performed in MPEG-4 decoding. For example, the frame detector (FD) models the part of the application that determines the frame type and the number of macro blocks to decode. The rest of the steps in MPEG-4 decoding are Variable Length Decoding (VLD), Inverse Discrete Cosine (IDC) Transformation, Motion Compensation (MC), and Reconstruction (RC) of the final video picture. □

The *SDF Metamodel* capturing the concepts of Definition 7.1 is shown in Figure 7.5. Recall that an SDF graph is a tuple $G = (A, D, Tok_0)$.

- *SynchronousDataflowRoot* is the root of a model, in which everything else is contained; it corresponds to G.
- *Actor* corresponds to the set A; the associations *incomingChannels* and *outgoingChannels* represent the derived functions *In* and *Out* from A to sets of channels.
- *Channel* corresponds to the set D. The 4-tuples $(a, b, p, q) \in D$ are represented in the metamodel by the *source* and *target* associations (for a and b), respectively the *sourceRate* and *targetRate* attributes (for p and

q). *initialTokens* represents the function Tok_0; thus, it has been modelled as an attribute of *Channel*, rather than as a separate function.

With respect to the mathematical definition, there are two differences: (i) whereas a channel (a, b, p, q) is completely determined by its constituent values, due to the nature of metamodels a *Channel* has its own identity (and so conceivably there could be two *Channel*s with the same 4-tuple of values), which can not occur in the mathematical set up in Definition 7.1; (ii) the function Tok_0 has been combined with *Channel*. This removes some of the modularity of the mathematical model, at the benefit of simplicity.

Model Creation

In our framework, SDF models are created in steps 1 and 2 of Figure 7.3. The starting point is an SDF graph created using the well-known open-source SDF^3 tool (step 1). This tool produces output in the form of an XML document, adhering to its own schema (fixed in an XSD). To bring such documents into our framework, we have defined an SDF^3*-to-SDF Transformation* which produces models conforming to the SDF metamodel of Figure 7.5. The transformation definition involves a systematic mapping of the SDF^3 concepts to our SDF metamodel concepts. The layout of XML format used in SDF^3 is already explained in Chapter 2.

Listing 7.1 shows a partial view of the SDF^3-to-SDF transformation. A rule in ETL starts with the rule name. Following the rule name, the `transform` keyword defines from which input elements to which output elements this rule does the mapping. The transformation direction is specified by the `to` keyword, which also acts as a separator between input and output elements. These elements are defined with a given name (that is used to refer to that element inside the rule body) followed by its type (in its metamodel) separated by ":". Following this, the rule body is defined inside curly brackets. The rules in Listing 7.1 can be explained as follows:

- *Sdf3Type* defines the SDF graph in the SDF^3 XSD schema. The first rule named *SDFGraph2SDFModel* at line 2 transforms an *Sdf3Type* element in the SDF^3 XML document to a *SynchronousDataflowRoot* element in the SDF model.
- *ActorType* defines an actor with a name in the SDF^3 XSD schema. The rule named *Actor2Actor* at line 7 transforms an *ActorType* element in the SDF^3 XML document to an *Actor* element in the SDF model. The rule sets the identifier property of the *Actor* element as the name property of *ActorType* element.
- *ChannelType* defines a channel with its properties in the SDF^3 XSD schema. The rule named *Channel2Channel* at line 13 transforms a *ChannelType* element in the SDF^3 XML document to a *Channel* element in the SDF model. At lines 16 and 17, the names and initial tokens of all channels in the SDF model are created according to the respective names and initial tokens in the SDF^3 XML document. At lines 18-22, the source actor and production rate of each channel in the SDF^3 XML document are assigned to

Listing 7.1: A partial view of SDF^3-to-SDF Transformation

```
1  /** Transforms an SDF3 root element to an SDF root element.*/
2  @rule@ SDFGraph2SDFModel transform
3    sdf3In: In!Sdf3Type to sdfOut: Out!SynchronousDataflowRoot
4  {
5  }
6  /** Transforms an SDF3 actor to an SDF actor.*/
7  @rule@ Actor2Actor transform
8    aIn: In!ActorType to aOut: Out!Actor
9  {
10   aOut.identifier = aIn.name;
11 }
12 /** Transforms an SDF3 channel to an SDF channel.*/
13 @rule@ Channel2Channel transform
14   cIn: In!ChannelType to cOut: Out!Channel
15 {
16   cOut.identifier = cIn.name;
17   cOut.initialTokens = cIn.initialTokens.asInteger();
18   // Assign source actor and production rate
19   var srcActorIn : In!ActorType = getInActor(cIn.srcActor);
20   cOut.source ::= srcActorIn;
21   cOut.sourceRate = srcActorIn.getInPort(cIn.srcPort).rate.
22       asInteger();
23   // Assign target actor and consumption rate
24   var dstActorIn : In!ActorType = getInActor(cIn.dstActor);
25   cOut.target ::= dstActorIn;
26   cOut.targetRate = dstActorIn.getInPort(cIn.dstPort).rate.
27       asInteger();
28   //Creating the list of incoming and outgoing channels of each
         actor
29   for (a in In!ActorType)
30   {
31     if (a.name == cIn.srcActor)
32     {
33       cOut.source.outgoingChannels.add(cOut);
34     }
35     else
36     {
37       cOut.target.incomingChannels.add(cOut);
38     }
39   }
40 }
```

corresponding channel in the SDF model, using the operations *getInActor* and *getInPort* (not shown here) respectively. Similarly, at lines 23-27, target actor and consumption rate of each channel in the SDF model are assigned. As mentioned earlier, each *Actor* in the SDF metamodel keeps the list of its incoming and outgoing channels. Therefore, at lines 28-39, each actor in the SDF^3 XML document is checked if it is the source or target actor of the current channel. If it is the source, then the channel is added as outgoing channel of the actor in the SDF model. Otherwise, the channel is added as the incoming channel.

7.4.2 Platform Application Models

A Platform Application Model (PAM) models the multiprocessor platform to which the application, modelled as an SDF graph, is mapped. Our PAMs support several features, including (1) heterogeneity, i.e., there can be multiple processors with different types, (2) a partitioning of the processors in voltage/frequency islands, (3) frequency levels each processor can run on, (4) power consumed by a processor at a certain frequency, both when in use and when idle, and (5) power overhead required to switch between frequency levels.

Definition and Metamodel

Definition 7.3. *Given an SDF graph* $G = (A, D, Tok_0)$ *with a set of actors* A*, a* platform application model *(PAM) is a tuple* $\mathcal{P} = (\Pi, \zeta, F, P_{idle}, P_{occ}, P_{tr}, \tau_{act})$ *consisting of*

- *a finite set of processors* $\Pi = \{\pi_1, \ldots, \pi_n\}$*. We assume that* Π *is partitioned into disjoint blocks of voltage/frequency islands (VFIs) such that* $\bigcup \Pi_i = \Pi$*, and* $\Pi_i \cap \Pi_j = \emptyset$ *for* $i \neq j$*,*
- *a function* $\zeta : \Pi \to 2^A$ *indicating which processors can handle which actors,*
- *a finite set* $F = \{f_1, \ldots, f_m\}$ *of discrete frequencies available to all processors,*
- *a function* $P_{occ} : \Pi \times F \to \mathbb{N}$ *denoting the power consumption (static plus dynamic) of a processor operating at a certain frequency* $f \in F$ *in the operating state,*
- *a function* $P_{idle} : \Pi \times F \to \mathbb{N}$ *denoting the power consumption (static) of a processor operating at a certain frequency* $f \in F$ *in the idle state,*
- *a partial function* $P_{tr} : \Pi \times F^2 \nrightarrow \mathbb{N}$ *denoting the transition overhead between frequencies for each processor* $\pi \in \Pi$*, and*
- *a function* $\tau_{act} : A \times F \to \mathbb{N}_{\geq 1}$ *denoting the actual execution time of each actor (in* A*) mapped to a processor at a certain frequency level (in* F*).*

As SDF graphs are considered to be untimed, the execution time of the SDF actors are defined along with PAM components. Moreover, we assume that the processors can switch from the active frequency only to the neighbouring frequency.

The principle of Dynamic Voltage and Frequency Scaling (DVFS) dictates that a processor can change its running frequency. A processor $\pi \in \Pi$ running at frequency $f \in F$ consumes $P_{occ}(\pi, f)$ if working, or $P_{idle}(\pi, f)$ otherwise. Intuitively, the lower the frequency $f \in F$ of a processor $\pi \in \Pi$ is, the lower the power consumption $P_{occ}(\pi, f)$ of that processor is, at the expense of a longer execution time $\tau(a, f)$ of an actor $a \in A$.

The state of a processor can be changed by Dynamic Power Management (DPM) at run-time when the processor is not in use. However, the power overhead

No.	Frequency(MHz)	$\mathbf{P_{idle}}$(W)	$\mathbf{P_{occ}}$(W)
1	1400	0.1	4.6
2	1033	0.4	1.8

Table 7.6: Platform description of Samsung Exynos 4210 processors

of switching to another power state (i.e., frequency level) is non-negligible. The power overhead is captured by the function P_{tr}.

Furthermore, the concept of voltage/frequency islands (VFIs) allows to run a group of processors at a common voltage/frequency. The clock frequencies/voltage supplies of a VFI may differ from other VFIs. Without VFIs, we are left with two options only, i.e., either running each processor at an individual voltage/frequency (local DVFS), or running all the processors at the same voltage/frequency (global DVFS). Hence, VFIs provide better control over energy management.

Example 7.4. Table 7.6 recalls two frequencies (MHz) $\{f_1, f_2\} \in F$ and corresponding experimental power consumption of Exynos 4210 processors from Chapter 5. We assume that our PAM contains four Exynos 4210 processors, i.e., $\Pi = \{\pi_1, \pi_2, \pi_3, \pi_4\}$. The processors are partitioned into two VFIs, i.e., $\Pi_1 = \{\pi_1, \pi_2\}$ and $\Pi_2 = \{\pi_3, \pi_4\}$. We assume that the power overhead (W) of all $\pi \in \Pi$ is, $P_{tr}(\pi, f_1, f_2) = 0.2$ and $P_{tr}(\pi, f_2, f_1) = 0.1$. □

The *PAM Metamodel* capturing most of the concepts of Definition 7.3 is shown in Figure 7.7. A brief explanation can be given as follows:

- *PlatformApplicationModelRoot* stands for the PAM as a whole.
- *ProcessorType* collects the characteristics of a set of processors. In the metamodel, the power and frequency characteristics of a processor are associated with its *type*, creating a reusable layer of indirection with respect to the mathematical model.
- *Processor* stands for the elements of Π. Each *Processor* has a *type* association to the corresponding *ProcessorType*.
- *VoltageFrequencyIsland* stands for the clusters Π_i in the VFI partitioning of Π. The element-of relationship between a processor and its VFI is captured by the (opposite) *island* and *processors* associations.
- *ProcessorState* associates the working/idle state of a processor (type) (the boolean *isWorking* attribute), combined with a *frequency* level, to a *powerConsumption* value. This encodes the P_{occ} and P_{idle} functions of the mathematical definition.
- *ProcessorStateChange* encodes the P_{tr} function of the definition: each instance associates a *powerCost* with a certain pair of *source* and *target* *ProcessorStates*.

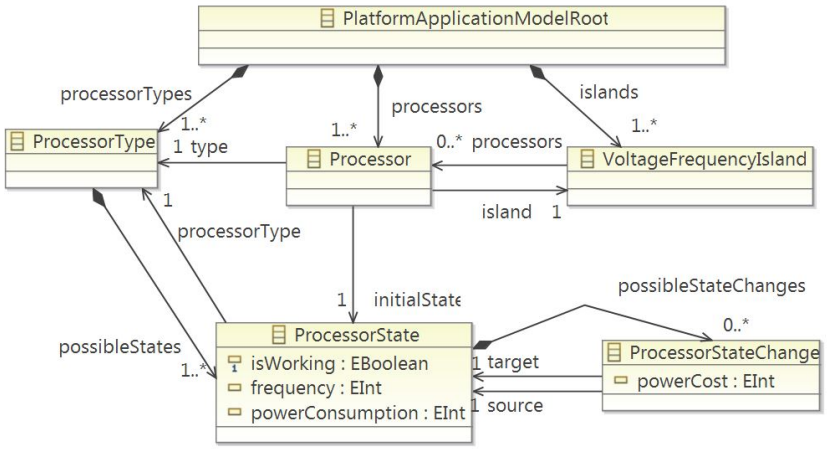


Figure 7.7: PAM Metamodel. *PlatformApplicationModelRoot* is the root of a model, *ProcessorType* corresponds to the processor types, *Processor* corresponds to the set of processors, and *VoltageFrequencyIsland* stands for the VFIs. Furthermore, *ProcessorState* and *ProcessorStateChange* encodes the states and transition overheads of the processor types respectively.

In a major change with respect to the mathematical definition, we have chosen not to include the ζ and τ_{act} functions in the PAM, but to isolate them in a separate allocation model. This enhances the modularity of the modelling framework. Apart from this change, all elements of Definition 7.3 are clearly recognisable in the metamodel, though sometimes encoded in a different manner. In particular, we have introduced the processor types as an intermediate level to enhance modularity; P_{occ} and P_{idle} are combined in *ProcessorState*; and P_{tr} is encoded as *ProcessorStateChange*.

Model Creation

The creation of PAMs corresponds to step 3 in Figure 7.3. Although EMF provides a default tree-based model editor, we have built PAM Visual Editor, a domain-specific visual editor for PAMs, by benefiting from state-of-the-art MDE techniques. To build PAM Visual Editor, we have used EuGENia, which can automatically generate a visual editor from an annotated ECore metamodel. We show an example PAM created using this visual editor in Section 7.5.

7.4.3 Allocation Models

In a heterogeneous system, the freedom of assigning actors $a \in A$ to processors $\pi \in \Pi$ is constrained by which processors can be utilised to execute a particular actor. Thus, in order to run an SDF model on a PAM, we need to know (1) which SDF actors can be run on which processors of the PAM and (2) what their execution times are at given frequencies. This information is encoded in

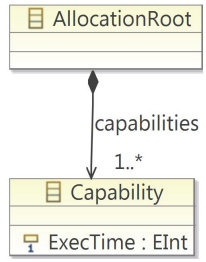


Figure 7.8: Allocation Metamodel. *Capability* refers to *Actor* in the SDF metamodel and *ProcessorState* in the PAM metamodel.

an allocation model, which relates both the SDF and PAM models. Allocation models conform to *Allocation Metamodel* that we define to represent this concern.

The information related to allocation concern was part of Definition 7.3, but we have chosen to put this into a separate *Allocation Metamodel* for the sake of modularity, making the PAM metamodel fully independent of the SDF metamodel.

Definition and Metamodel

The information to be represented in the Allocation metamodel consists of the ζ and τ_{act} functions of Definition 7.3. The *Allocation Metamodel* is shown in Figure 7.8. It contains:

- *AllocationRoot*, which stands for the combined allocation functions ζ and τ_{act} of Definition 7.3.
- *Capability*, following $\tau_{act} : A \times F \to \mathbb{N}_{\geq 1}$ in Definition 7.3, refers to *Actor* in the SDF metamodel, and *ProcessorState* (defining the frequency of the processor) in the PAM metamodel, and yields the time needed to execute the actor at the processor state. At the same time, *ProcessorState* also encodes which processor type an actor can be executed on.

The metamodel is in fact more expressive than the mathematical definition: for instance, the execution time of an actor is not constrained to be always the same for a given frequency level; instead, it may also depend on the processor type.

Model Creation

The creation of Allocation models corresponds to step 4 in Figure 7.3. It is supported out-of-the-box via the default tree-based model editor provided by EMF.

7.4.4 Common Metamodel

In addition to those discussed above, Figure 7.3 also shows an element called *Common Metamodel*. This demonstrates an MDE technique for reuse: our

common metamodel defines the general concept of *Identifiable*, which has a string-valued *identifier* attribute; *Actor*s and *Processor Type*s are subtypes of *Identifiable* and thereby inherit this feature. Whenever (during extension of the framework) additional reusable concepts are introduced, these can be added to the common metamodel.

7.4.5 Priced Timed Automata Models

Once the SDF, PAM and allocation models are available, one can generate the PTA model using the *Co-Design-to-*UPPAAL *Transformation* and successive model-to-text transformation. These correspond to steps 5 and 6 in Figure 7.3. Step 7 in Figure 7.3 corresponds to reusing the idea in Chapter 5, to generate the energy-optimal schedule of an SDF graph.

Co-Design-to-UPPAAL transformation

Given an SDF graph $G = (A, D, \mathsf{Tok}_0)$ mapped on a processor application model $\mathcal{P} = (\Pi, \zeta, F, P_{idle}, P_{occ}, P_{tr}, \tau_{act})$, the Co-Design-to-UPPAAL transformation creates a parallel composition of PTA:

$$A_G \| Processor_1 \| , \ldots , \| Processor_n \| Scheduler.$$

Here, the automaton A_G models the actors and channels of an SDF graph, and the automata $Processor_1, \ldots, Processor_n$ model the processors $\Pi = \{\pi_1, \ldots, \pi_n\}$. The automaton *Scheduler* is responsible for the scheduling task.

This parallel composition of automata is represented as the PTA model generated as the output of the Co-Design-to-UPPAAL transformation. This model conforms to the UPPAAL metamodel. The metamodel not only contains the conceptual timed automaton elements such as locations, edges and clocks; but also specifies the abstract syntax of the plain-text expressions. PTA models are already explained in Chapter 5.

Model-to-text transformation

As mentioned earlier, the model generated by step 5 does not have the right format to be directly processable by UPPAAL CORA. For this reason, we have defined a model-to-text transformation, which takes an UPPAAL CORA model as input and transforms it into the native UPPAAL CORA XML format. This transformation corresponds to step 6 in Figure 7.3.

Computation of the Energy-Optimal Schedule

This activity corresponds to step 7 in Figure 7.3. We reuse the idea presented in Chapter 5, to determine the throughput of an SDF graph, and energy optimisation with respect to it. The trace generated by UPPAAL CORA is interpreted to derive an energy-optimal schedule.

7.5 Case Study and Evaluation

In this section, we show the effectiveness of our framework for HW-SW co-design by applying it on a case study. We also demonstrate how our framework satisfies the features stated in Section 7.1, namely: (1) modularity, (2) extensibility, and (3) interoperability. We also evaluate the timing performance of our framework with the help of some other case studies.

7.5.1 Case Study

As a case study, we consider the dataflow application of the MPEG-4 decoder in Figure 7.4 on page 161 mapped to a platform with four processors of the type Exynos 4210 in Example 7.4 on page 165.

Following step 1 in Figure 7.3, we have created the SDF graph of the MPEG-4 decoder using SDF^3. This SDF graph was already given in Figure 7.4. In step 2, we apply SDF^3-to-SDF transformation to generate the SDF model conforming to our metamodel for the SDF graph.

In step 3, we create the PAM using the visual editor, as described in Section 7.4.2. The created PAM is given in Figure 7.9. The top part of the figure shows four processors partitioned into two VFIs. The big rectangle at the bottom part of the figure shows the processor type, which is Samsung Exynos 4210 for our case. The oval shapes inside the processor type represent the processor states. For example, at the frequency level (MHz) $f_1 = 1400$, a processor $\pi \in \Pi$ consumes $P_{idle}(\pi, f_1) = 0.1$ W if it is idle, or $P_{occ}(\pi, f_1) = 4.6$ W if it is working. The arrows between processor states represent transitions between states, and the power overhead of each transition are assigned to the respective arrows. For example, as $P_{tr}(\pi, f_1, f_2) = 0.2$ W, the arrow pointing from State 2 (representing the idle processor at f_1) to State 4 (representing the idle processor at f_2) has value 2 assigned to it. As UPPAAL CORA only can have integer costs, all power consumption values in Figure 7.9 are multiplied by 10. This can also be done in the Co-Design-to-UPPAAL transformation in step 5.

After we have the PAM and SDF models, we create the allocation model that assigns the actors to the processor states with their execution times in step 4.

Once we have the SDF, PAM, and allocation models, we apply the Co-Design-to-UPPAAL transformation in step 5 and the model-to-text transformation in step 6 to generate the PTA model that is compatible with UPPAAL CORA. The UPPAAL CORA models of our running example are already presented in Chapter 5.

In step 7, we follow the approach presented in Chapter 5 to compute the energy-optimal schedule for some given throughput requirements.

Figure 7.10 shows the energy-optimal schedule, for the time per graph iteration constraint of 8 ms for our example. The schedule shows the execution order of the actors at the specific frequency and processors.

7.5.2 Evaluation

a) Modularity:. To show that our framework satisfies the modularity criterion, let us consider the following example: We want to analyse the energy consumption

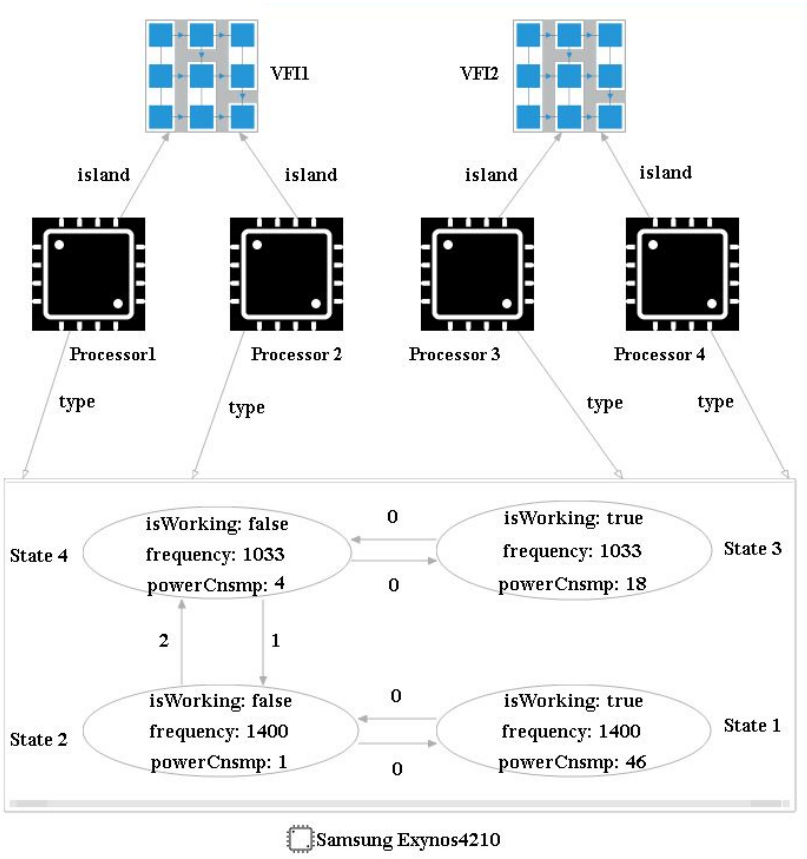


Figure 7.9: PAM created using PAM Visual Editor. The top part shows four processors partitioned into two VFIs. The lower rectangle shows the processor states and transition overheads of the processor type Samsung Exynos 4210.

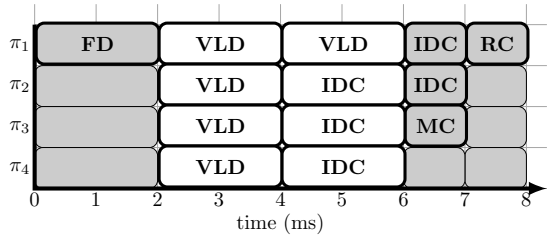


Figure 7.10: An example schedule of MPEG-4 decoder in Figure 7.4 on four processors π_1, π_2, π_3, and π_4. The grey and white coloured boxes denote, if a processor is running at the frequency f_2 or f_1 respectively.

of the MPEG-4 decoder on a different hardware platform, viz., Intel Core2 Duo E6850.

No.	Frequency(MHz)	$\mathbf{P_{idle}}$(W)	$\mathbf{P_{occ}}$(W)
1	3006	0.1	5.5
2	1776	0.4	2.2

Table 7.11: Platform description of Intel Core2 Duo E6850

Table 7.11 shows the frequencies and corresponding power consumption of this new processor [PPS+13]. For this scenario, we only changed the processor type while keeping the number of processors and VFI distributions the same. Now, all we have to do is to develop a new PAM corresponding to this new platform specification and generate the corresponding PTA model. We reuse the existing SDF model of the application without making any modifications. Using the framework, we derive the energy-optimal schedule for time per graph iteration constraint of 8 ms. The schedule remains the same as Figure 7.10.

c) Interoperability:. As explained in Section 7.1, HW-SW co-designing involves tools having different domains and focus areas. There must be interoperability between these tools to work together efficiently. In our framework, we utilise SDF^3 for creating SDF graphs and UPPAAL CORA for deriving energy-optimal schedule. To automatically generate UPPAAL CORA models from SDF^3 models, we have implemented model transformations in our framework, thus providing interoperability.

b) Extensibility:. Due to the adaptation of MDE techniques, our framework provides systematic extensions for future requirements. As an example, suppose we want to extend our hardware platform models with the concept of "data memory". The current version of the hardware platform does not consider memory access times of processors, which means processors can access data without incurring any read or write time. However, we want to include a data memory in our HW-SW co-design as a resource. For this purpose, we extend Definition 7.3 of PAM to $\mathcal{P} = (\Pi, \zeta, F, P_{idle}, P_{occ}, P_{tr}, \tau_{act}, DM, Rd_{DM}, Wr_{DM})$. The new components are defined as follows:

- a global data memory element DM,
- a function $Rd_{DM} : DM \to \mathbb{N}$ indicating the read time of the data memory DM, and
- a function $Wr_{DM} : DM \to \mathbb{N}$ indicating the write time of the data memory DM.

Figure 7.12 shows the execution phases of an actor $a \in A$ on a processor $\pi \in \Pi$ at a frequency $f_i \in F$ considering the memory accesses. The execution starts with a read phase. The processor $\pi \in \Pi$ reads the memory DM to access tokens on all input channels of the actor a. After the read phase, the processor

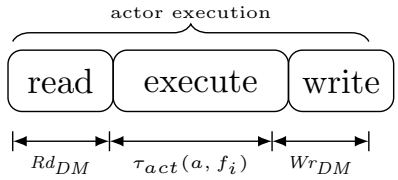


Figure 7.12: Execution phases of an actor a at a frequency f_i. First the input tokens are read from the memory, and then the actor is executed and the output tokens are produced. Afterwards , the output tokens are written on the memory. The whole process takes $Rd_{DM} + \tau_{act}(a, f_i) + Wr_{DM}$ time units.

π executes the actor a in $\tau_{act}(a, f_i)$ time units. Finally, in the write phase, the produced tokens are written to the memory DM.

To extend our framework to support the memory concept, we perform the following steps:

- *Extending the PAM metamodel*: We define a *Memory Metamodel* as an extension of the PAM metamodel, as shown in Figure 7.13. The Memory metamodel includes only a single element called *Memory*. Memory is a singleton entity with the *readRate* and *writeRate* attributes that represents the time cost of reading and writing from the memory, respectively. Memory has also the *root* reference to the root element of the PAM metamodel.

- *Extending the Co-Design-to-*UPPAAL *model transformation*: To reflect the memory access times in the scheduling analysis, we have to adjust the Co-Design-to-UPPAAL transformation accordingly. This is achieved in the following way: The memory access times should be added to the execution time of each actor. The execution times of the actors are mapped to the invariants of the locations (that correspond to the working state of processor types) in PTA models. This mapping is implemented in the *PAMProcessorState2InUseLocation* transformation rule. We define a new transformation called *Memory_Extension Transformation* that extends the Co-Design-to-UPPAAL transformation and overrides the *PAMProcessor-State2InUseLocation* transformation rule. This is shown in Figure 7.14. In this way, we only change the single rule that is relevant to our purpose and reuse the rest of the original transformation.

7.5.3 Timing Performance

To determine the timing performance of our framework, we consider four real-life case studies namely, an MPEG-4 Decoder in Figure 7.4, an MP3 Decoder in Figure 2.15, an MP3 playback application in Figure 2.16, and an Audio Echo Canceller in Figure 2.17. We also used an artificial bipartite SDF graph with 4 actors in Figure 2.18. We assume that these case studies are mapped on Exynos 4210 processors having two frequencies.

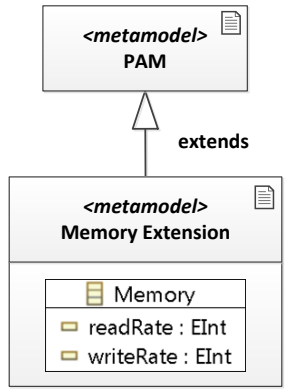


Figure 7.13: Memory Metamodel as an extension of PAM Metamodel. *readRate* and *writeRate* represents the time cost of reading and writing from the memory, respectively.

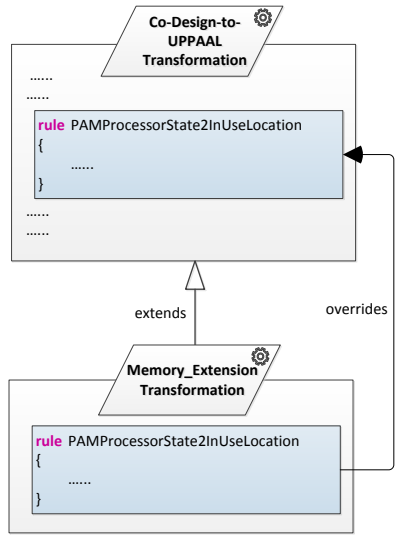


Figure 7.14: Extension of the Co-Design-to-UPPAAL Transformation. The time cost of reading and writing from the memory is added to the execution times of each actor in the PTA model of *Processor*.

We examine the timing performance of our framework in two parts: the first part is the timing performance of our framework, i.e., cumulative computation time of steps 2 (SDF^3-to-SDF transformation), 5 and 6 (Co-Design-to-UPPAAL and model-to-text transformations). The second part is the timing performance of obtaining the optimal schedule via UPPAAL CORA model checker, i.e., step 7.

Number of Processors	Steps 2+5+6	Time per Iteration	Energy Consumption	Step 7
MPEG-4 Decoder in Figure 7.4				
5	0.253	6	62.8	18826.46
4	0.253	6	64.6	47.03
3	0.145	7	64.3	3.33
2	0.142	9	64	0.41
1	0.140	14	64.4	0.21
MP3 Decoder in Figure 2.15				
2	0.281	8	64.6	1.41
1	0.256	14	64.4	0.21
MP3 Playback Application in Figure 2.16				
2	0.101	1880	9907	28800.05
1	0.081	2118	9742.8	71.15
Audio Echo Canceller in Figure 2.17				
4	0.119	23	324.2	13.78
3	0.108	24	322.3	0.71
2	0.106	35	322	0.71
1	0.108	73	335.8	0.41
Bipartite Graph in Figure 2.18				
4	0.123	42	345.3	366.04
3	0.110	44	338.5	145.12
2	0.102	51	333.1	4.95
1	0.102	73	335.8	0.71

Table 7.15: Timing performance of our framework for different case studies

The experiments were conducted on a dual-core 2.8 GHz machine with 8 GB RAM.

Table 7.15 shows the results of the experiments. The first column shows the number of processors available for the experiment. The second column shows the timing performance (s) of the first part. The columns 3-5 show the timing performance of the second part. For time per graph iteration constraint (ms) in column 3, the optimal energy consumption (mWs) derived by UPPAAL CORA is shown in column 4, and column 5 shows the time (s) taken to compute the results in column 4. As one can realise, the time that step 2, 5 and 6 take in total does increase insignificantly as the number of available processor increases. This is due to the slight increase in the model size with the addition of processor instances. For step 7, the time required to complete increases exponentially as

the number of processor increases, which is because of the fact that the size of the state-space created by the model checker increases exponentially with the size of the model itself.

7.6 Conclusions

In this chapter, we have presented a model-driven framework for HW-SW co-design of dataflow applications. In our framework, we have proposed a reusable set of three coherent metamodels for HW-SW co-design domain. To provide interoperability among domains, we have defined a reusable set of extensible model transformations. We have demonstrated that our framework satisfies the modularity, interoperability and extensibility requirements with a case study.

As future direction of our work, we plan to extend our framework with other analysis techniques such as simulation and automated HW-SW partitioning. We also plan to add code generation functionality to our framework.

Case Study: A Face Recognition System

Abstract

THE proof of the pudding is in the eating. The main research question of this chapter is to evaluate the performance of our methods in a real-life case study. In particular, we are interested in comparing theoretical and actual results, which will help us in understanding the effect of communication overheads and context switching present in real-life hardware platforms. To address this question, we consider the approach of deriving schedules of an SDF graph on a limited number of processors presented in Chapter 4. For this purpose, the STARS tool-chain discussed in Chapter 4 is utilised. The case study is a concrete case from Recore Systems, The Netherlands, an industrial partner in the European project SENSATION. The (theoretical) schedules obtained using our method have been implemented in the case study to determine the (actual) throughput. The experimental evaluation shows that the final deviation of the theoretical results from the actual results is 28% (see Figure 8.8) which can be attributed to communication overheads and context switching. Furthermore, the scalability evaluation shows that the run time of our approach exceeds 8 hours if there is more than one processor. We tackle this issue by utilising the random optimal depth first search option in UPPAAL that searches the successor states in a random fashion and generates a best trace out of all witnessed traces.

8.1 Introduction

Face Recognition System. Face recognition is an embedded multimedia application which identifies or verifies a person from a digital image. This is done by comparing facial features from the image and a facial database. Face recognition systems are getting huge popularity in application areas like surveillance, biometric security, video games, and image messaging applications. All of these application areas are constrained to adhere to stringent completion times. For example, police cannot afford to spend too much time in identifying a suspect from the CCTV footage. Similarly, facial recognition of a user connected to its avatar in Xbox's Avatar Kinect must be done with minimum delay.

Proposed Approach and Contributions. All this makes the face recognition system a perfect case study to investigate the applicability of the theory and analysis techniques presented in this thesis. In particular, we utilise the approach presented in Chapter 4 of optimal scheduling of dataflow applications on a given number of processors. This approach is supported by the STARS tool-chain which is also presented in Chapter 4. The case study of face recognition system is provided by the company Recore Systems [REC].

However, the STARS tool-chain does not consider any communication costs, and runtime synchronisation and updates to the administration. Thus, it would be relevant to evaluate how well the schedules obtained using the STARS tool-chain perform if implemented on a realistic hardware platform.

To achieve this goal, the first step is to obtain the SDF graph of the face recognition system application. This step is done by executing the application on a hardware platform. As a result of this step, a trace file is generated by which we determine the firing times and order of tasks which we use to extract an SDF graph.

The second step is to utilise the STARS tool-chain to generate (theoretical) schedules showing mappings of SDF actors to processors. Afterwards, the face recognition system application is executed on an actual hardware platform but considering the mappings generated by the STARS tool-chain in the last step. This gives us the (actual) schedules, which we compare to see the performance of our method.

The main contributions of this chapter are twofold.

- First, we evaluate how our method performs in a real-life large scale application. This helps to understand how much abstracting away from the communication costs affects. Moreover, we also evaluate the scalability of our method.

- Second, we determine the behaviour of the case study such as speedup ratio when increasing the number of processors.

Origins of the Chapter. The case study of face recognition system has been kindly provided by Recore Systems, The Netherlands, who were an industrial partner with University of Twente in an EU FP7 project SENSATION.

Chapter Outline. Section 8.2 explains the case study, and Section 8.3 presents the research questions posed in this chapter. The experimental setup used for addressing the research questions is illustrated in Section 8.4, and Section 8.5 discusses the results. Finally, Section 8.6 draws conclusions.

8.2 Description of the Case Study

This section describes face recognition systems, and the hardware platform developed by Recore Systems termed FLEXAWARE.

8.2.1 Application: A Face Recognition System

A face recognition system is an embedded multimedia application for automated identification of a person from a digital image. Recently, face recognition systems are being integrated more and more into embedded multimedia systems. For example, face recognition based games such as NBA 2K17 in Xbox 360 [NBA16], and mobile applications such as Oasis [OAS15] for unlocking mobile phones or mobile banking using face recognition. Similarly, face recognition is being used nowadays for better shopping by taking a photo of something, e.g., a book, and mobile applications like camfind [CAM13] compares price in multiple stores. In this case, instead of face, the features of a book such as title or ISBN are detected and identified. Other than embedded multimedia systems, the application areas of face recognition systems are video surveillance, CCTV control, suspect tracking etc.

A typical face recognition system consists of the following two steps, as shown in Figure 8.1.

1. face detection, and

2. face recognition.

Each step is briefly discussed in the following.

Face Detection

Face detection starts with image acquisition either by digitally scanning an existing photograph or by using a camera to capture a live photograph of a person. The next phase is to locate human faces in an image. The most commonly used face detection algorithm termed *Viola-Jones face detector* was introduced by Paul Viola and Michael Jones in 2001 [VJ01]. In this algorithm, the pixel intensities (RGB values) of different features of a human face such as eyes, nose etc. are learned by a *classifier*. Afterwards, the classifier is moved over the input image in the form of a search window. For each subsection of the image, the pixel intensities are calculated and compared to the learned pixel intensities. In this way, the face is identified from the other things in the image such a buildings, trees, bodies etc.

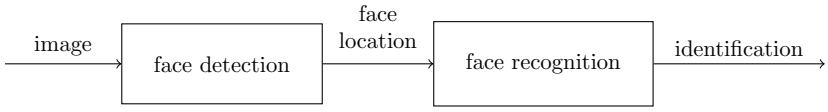


Figure 8.1: An overview of a face recognition system. The first step is of identifying and locating a face in an input image. The second step is of recognising a face with the aid of a facial database.

Face Recognition

After the face has been detected and extracted from an image, the next step is to compare it with the facial database and recognise it. To realise this step, a *feature vector* is constructed. The simplest form of a feature vector are the pixel intensities of different features of a human face calculated in the last step. A feature vector is a numerical representation of a human face. Similarly, each face in the database is also represented by a feature vector. Having such representation facilitates comparison, as well as statistical analysis of an image.

After constructing the feature vector of the detected face, it is compared with all feature vectors in the database to calculate a *score* vector. The best score is compared to a threshold to verify if the input image is in the database.

8.2.2 Platform: FLEXAWARE

Recore Systems has developed a face recognition system application on top of its FLEXAWARE platform [FLE] consisting of the following components.

- A heterogeneous many-core hardware platform containing embedded processors, digital signal processors, and function accelerators. To enable communication between processing cores, memories and interfaces, a dedicated on-chip interconnect is integrated in the many-core hardware architecture.

- A many-core operating system termed FLEXAWARE *runtime* that exposes an application programming interface (API) to run user applications on the platform.

FLEXAWARE also offers an intuitive software development environment (SDE) to provide a convenient implementation of an application on the FLEXAWARE platform. Using the FLEXAWARE SDE, user models the applications into sets of parallel tasks. Afterwards, the mapping of tasks to processors and memory allocation is done using the FLEXAWARE runtime under the hood.

At the time of writing, the FLEXAWARE platform of Recore Systems is still in the development phase, and the FLEXAWARE runtime experiments are run on a x86_64 architecture containing four processors.

8.3 Research Questions

The central research questions addressed in this chapter are as follows.

- How much the theoretical results calculated using the STARS tool-chain deviate from the results measured in the actual case study.

- How well our approach presented in Chapter 4 scales in an industrial case study.

- What is the speedup ratio of the case study when the number of processors is increased.

8.4 Experimental Setup

This section first presents an overview of the experimental setup considered in this chapter. Then, a detailed explanation of the experimental setup is given.

8.4.1 Overview of the Experimental Setup

The experimental setup consists of the following two stages.

1. An SDF graph of the face recognition system application is obtained by running the application on a single processor using the FLEXAWARE runtime. As a result, a trace file is generated from which we acquire the firing times and order of tasks of the face recognition system application. By utilising this information in the trace file, an SDF graph of the application is obtained.

2. The SDF graph obtained in the last stage is analysed using the STARS toolchain to generate (theoretical) schedules on a varying number of processors. The mappings in these schedules are fed to the FLEXAWARE runtime to derive (actual) schedules. Afterwards, the difference between theoretical and actual schedules is determined.

8.4.2 Details of the Experimental Setup

Figure 8.2 shows a detailed overview of the experimental setup.

Compilation and execution using FLEXAWARE Runtime

1. The first step is to extract an SDF graph of the face recognition system application in the SDF^3 XML format. The input to this step is the application which is compiled and executed on the single threaded FLEXAWARE runtime. The output of this step is a trace file in the FLEXAWARE trace format (FTF). The FTF file contains the firing times and communication topology of tasks of the face recognition system application.

   In our experiment, the application was executed 20 times. A single execution took an input stream of 10 video frames. The facial database contained 2048 images, and the face recognised in the input frame was compared to every image in the database to determine the best match.

SDF graph extraction using Host PC

2. The FTF file produced in step 1 is an input to step 2, in which the firing times and communication topology of tasks in the FTF file are parsed by the Host PC. The firing times of tasks are used to determine the execution times of SDF actors, and communication topology is used to build SDF actors and dependencies between them. This information is utilised to generate an SDF graph of the application as an output of step 2.

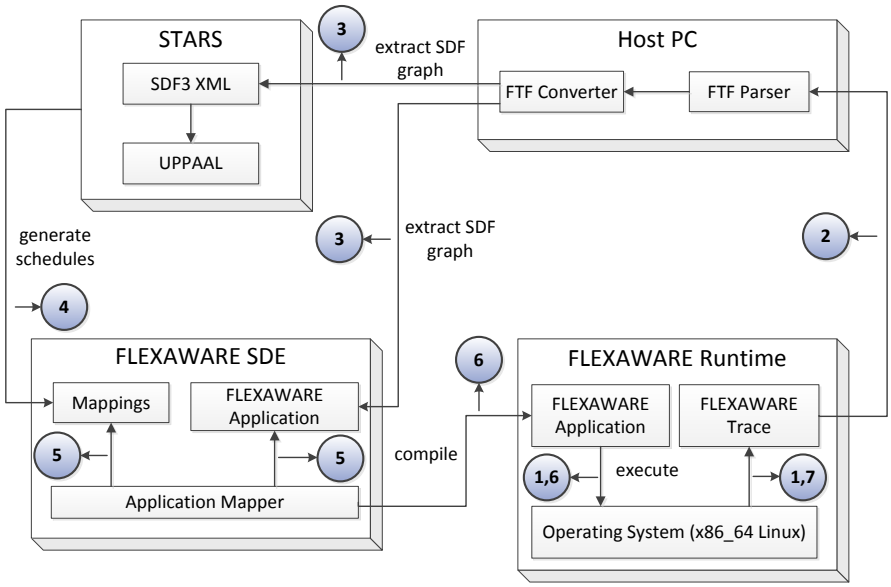


Figure 8.2: Experimental setup used for the analysis of face recognition system. The numbered steps are explained in the main text.

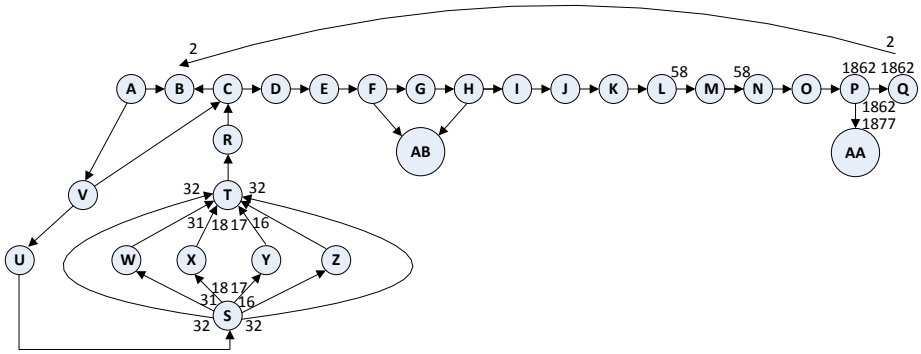


Figure 8.3: SDF graph of face recognition system

Figure 8.3 shows the anonymised SDF graph of the face recognition system consisting of 28 actors. For the sake of readability, production/consumption rates equal to one and initial tokens are omitted. Furthermore, in Figure 8.3, the degree of auto-concurrency of all actors is one (hence, we omitted the self-loop with rate one for all actors for simplicity).

As mentioned earlier, the application was run for 10 times in our experiment. The execution times of actors are calculated by taking average of the resource usage by each actor in each execution. Hence, the considered

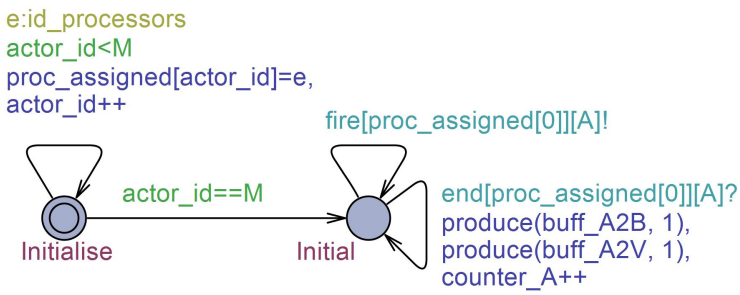


Figure 8.4: UPPAAL model for SDF graph in Figure 8.3 with respect to actor A. The UPPAAL model is modified so that an actor can run on one processor only throughout the execution.

execution times are not worst-case.

3. The SDF graph obtained in step 2 is an input to step 3, in which an SDF^3 XML file is extracted by an FTF Converter as an output.

Generation of schedules using STARS tool-chain

4. Step 4 takes as an input the SDF^3 XML file generated in step 3, and translates it to the UPPAAL model checker which generates schedules on the varying number of processors, as explained in Chapter 4. In this step, we choose the number of processors to vary from one to four.

   Restrictions in the UPPAAL model. We have an extra constraint in the UPPAAL model that an actor should always fire on the same processor during the entire execution. This is done to avoid costless task migration. To integrate this constraint, we have to modify the UPPAAL models explained in Chapter 4. Figure 8.4 shows the modified UPPAAL model for the SDF graph in Figure 8.3 with respect to actor A. Please note that Figure 8.4 only shows the SDF graph template; the PAM template remains unaffected.

   As seen in Figure 8.4, we have made the following modifications to the UPPAAL models explained in Chapter 4.

   - an integer variable proc_assigned[M] where M is the total number of actors in the SDF graph. In our case, the value of M is 28.
   - an integer variable actor_id with an initial value of zero. This variable is used to allocate a processor to each actor on which that actor fires for the whole execution. Recall that each actor in the SDF graph is assigned an id in the range [0,M-1].
   - a location Initialise with a self-edge.
   - an edge from the location Initialise to the location Initial.

The SDF graph template starts in the location Initialise, and takes the self-edge on the location Initialise. The self-edge has a label e : id_r which selects a processor id nondeterministically from user-defined type id_r and stores in e. As a result of taking this edge, the processor id stored in e is assigned to actor_id via the assignment proc_assigned[actor_id] = e. Moreover, the value of actor_id is incremented by one. This edge can be taken before the value of actor_id turns to M, annotated by the guard condition *actor_id*<M. In this way, starting from the actor with id equal to zero, a processor is assigned to each actor.

The edge from the location Initialise to the location Initial is taken when the value of *actor_id* is equal to M, specifying that each actor is assigned a processor to map on. The self-edges originating from the location Initial has one modification. Originally, the edges are annotated with a *Select* label e : id_r which selects processor ids nondeterministically, and the actors are mapped to the processors using the actions fire[e][actor_id] and end[e][actor_id]. Now, the processors are already assigned, therefore we omit the *Select* label. Rather, the actors are mapped on the assigned processors using the actions fire[proc_assigned[actor_id]][actor_id] and end[proc_assigned[actor_id]][actor_id]. For example, in Figure 8.4, the actor *A* having an id equal to zero, maps to its assigned processor using the actions fire[proc_assigned[0]][A] and end[proc_assigned[0]][A].

Hardware constraints specification using FLEXAWARE SDE

5. After schedules are generated, the next step is to validate them. Step 5 takes as an input the schedules generated in step 4, and specifies the hardware constraints, i.e., deployment of an actor to its assigned processor in the corresponding schedule generated in step 4. This is done by the *application mapper* in the FLEXAWARE SDE which creates a *mapping*. This mapping is stored as a mapping constraint file which is also an output of step 5.

Measurements using FLEXAWARE runtime

6. Step 6 takes as an input a mapping constraint file produced in step 5. Furthermore, in step 6, the face recognition system application is compiled and loaded in the FLEXAWARE runtime. During the application execution, the creation of tasks is handled by the *runtime service* shown in Figure 8.5. The FLEXAWARE runtime assigns actors to the runtime kernel corresponding to the processor according to the mapping constraint file constructed in step 5. As a result of step 6, a trace file is generated. This trace file contains per processor, the start and end times of intervals during which a specified actor has been active on that processor.

7. In step 7, the trace file constructed in step 6 is translated to a schedule.

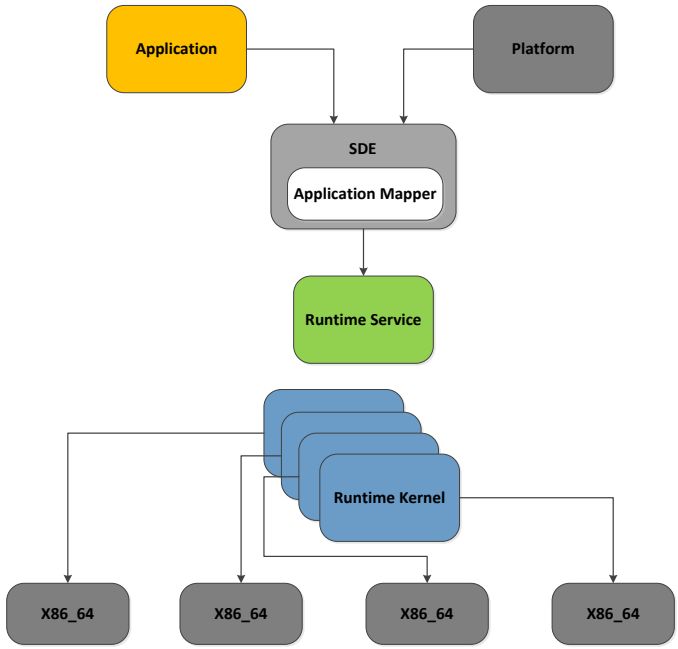


Figure 8.5: Mapping constraints supported by FLEXAWARE runtime

8.5 Results

In this section, we first evaluate the performance of our approach by comparing the calculated and measured results. Next, we discuss the speedup of the face recognition application system based on the measured results. Last, we evaluate the scalability of our approach in terms of computation times and memory consumption.

8.5.1 Performance Evaluation

Table 8.6 compares the completion times (μs) calculated by the STARS tool-chain and measured by FLEXAWARE. Column 1 shows the number of frames processed by the face recognition system. Column 2-5 shows the completion times calculated by the STARS tool-chain on a varying number of processors, and columns 6-9 presents the completion times measured by FLEXAWARE. The same results are plotted in Figure 8.7.

We can see in Table 8.6 and Figure 8.7 that increasing the number of processors leads to higher speedup, both in calculated and measured completion times. However, there is an exception in the case of four processors, where the measured completion times are lower than three processors. In fact, the measured completion times for four processors are closer to two processors.

The reason is probably that the considered platform consists of four pro-

Number of Frames	Calculated Completion Time (Number of Processors)				Measured Completion Time (Number of Processors)			
	1	2	3	4	1	2	3	4
1	566735	438743	359237	375367	433059	436358	408097	443019
2	1133470	750976	595183	587446	1609664	1166961	1015100	1420824
3	1700200	1047510	878083	807429	2227867	1587763	1278901	1820026
4	2266940	1349500	1146090	1076570	2844070	2005365	1580103	2214227
5	2833680	1669730	1397580	1247400	3458672	2423167	1868905	2603030
6	3400410	1937980	1678160	1467380	4077275	2851169	2141905	2993231
7	3967140	2303460	1894070	1692310	4692878	3267770	2420907	3389033
8	4533880	2572960	2228080	1932930	5313280	3690173	2708308	3773835
9	5100620	2919420	2529470	2161740	5920884 3	4108174	3000509	4148036
10	5667350	3233340	2803290	2347310	6495886	4492376	3230110	4242837

Table 8.6: Comparison of calculated (STARS) and measured (FLEXAWARE) completion time (μs) for varying number of frames and processors. The leftmost column shows the number of frames processed by face recognition system. The number of processors are shown in the second row.

cessors, and when the schedules are mapped on four processors, there is no room for operating system to perform administrative tasks and routines, e.g, updating the internal operating system kernel clock. Thus, because of interference from the operating system, the progress of the main application is involuntarily interrupted. As a result, performance is degraded on four processors compared to three processors. In the rest of the chapter, the analysis for four processors is neglected.

Figure 8.8 shows the difference between the calculated and measured completion times, where we see three trends. A reasonable explanation of these trends is discussed below.

- At the first frame, the measured completion times are lower than the calculated completion times. The reason is that the execution times of SDF actors are an average over the run of 10 frames. At the first frame, the execution times of the SDF actors are too pessimistic. Furthermore, for processing one frame, the buffers are empty and internal actions never get blocked. Therefore, the first frame completes earlier than expected.

- At the second frame, the measured completion times are higher than the calculated completion times. This is due to the fact that at the second

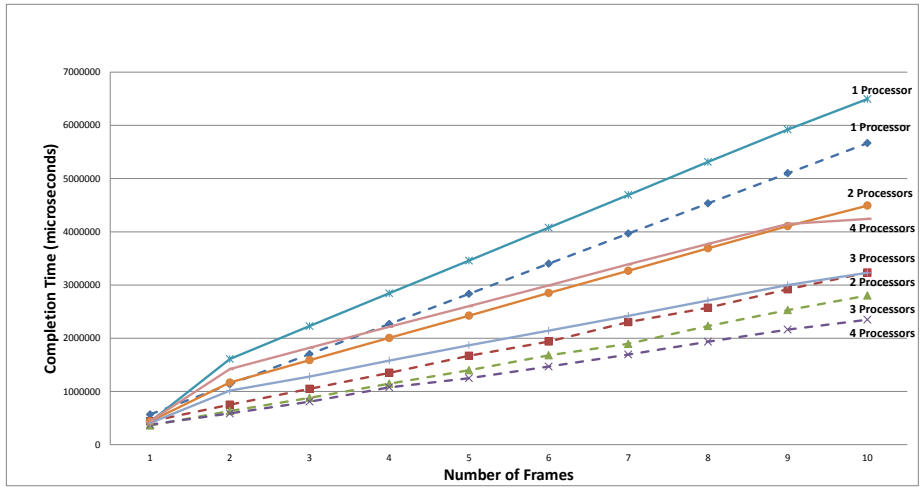


Figure 8.7: Comparison of calculated (STARS) and measured (FLEXAWARE) completion times (μs) for varying number of frames and processors. The dashed and solid lines represent the calculated and measured completion times respectively.

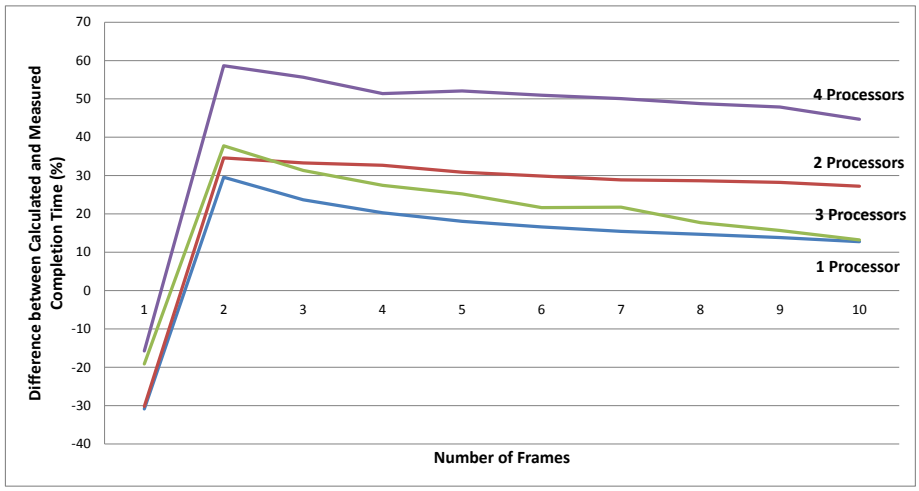


Figure 8.8: Difference between calculated (STARS) and measured (FLEXAWARE) completion times for varying number of frames and processors

frame, the *pipeline* is not sufficiently filled, and during the idle slots, there is not any other frame in the pipeline to be started.

- From the third frame onward, the difference between the calculated and measured completion times starts reducing. This is caused due to sufficient

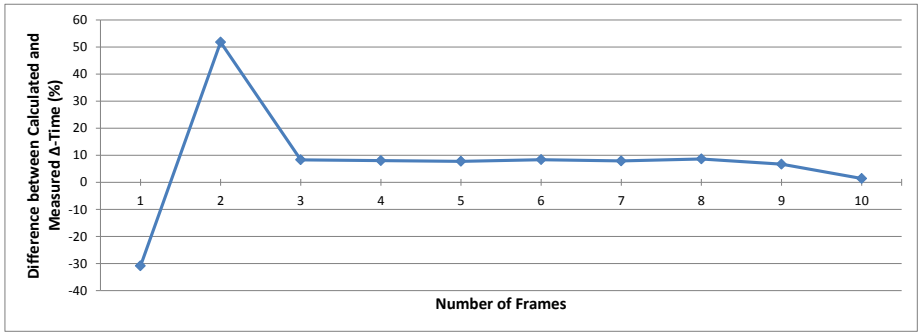


Figure 8.9: Δ-time difference between calculated (STARS) and measured (FLEXAWARE) completion times for one processor and varying number of frames

filling of the pipeline. Thus, the idle slots are better utilised by executing next frames, even though the previous one(s) are still executing. Furthermore, the observed execution time of a task drops below the average time of the SDF actor, as the initial penalty of the insufficient pipeline filling becomes insignificant.

The results in Figure 8.8 can be seen as "transient" and "steady-state" phases. The transient phase consists of the first and second frame, whereas the steady-state phase comprises of the third frame and onward. If we ignore the results for four processors, the deviation of the calculated results from the measured results at the 10^{th} frame is 28% (for 2 processors).

Discussion

The first frame completes earlier than expected on one processor, due to pessimistic execution times. The second frame takes much longer to complete, because of insufficient pipeline filling. In the steady-state phase, the difference starts decreasing. At the 10^{th} frame, the difference between the calculated and measured results is 13% for 1 and 3 processors, and 28% for 2 processors. This difference can be associated to communication overheads and context switching. Considering the fact that the extracted SDF model —based on runs on a single processor— does not contain overheads, this difference is in line with the expectation.

The transient and steady-phase behaviour of the case study can be better understood by looking at the Δ-times. For i^{th} frame, the $\Delta\text{-}time(i)$ is defined as, $Completion\ time(i) - Completion\ time(i-1)$, where $Completion\ time(0) = 0$. Figure 8.9 shows the Δ-time difference between the calculated and measured completions times for one processor and varying number of frames. We can see the same trend in Figure 8.9 as Figure 8.8. For the first frame, the Δ-time difference is negative, as the actual execution times are lower than the average execution times. For the second frame, the Δ-time difference is positive. After

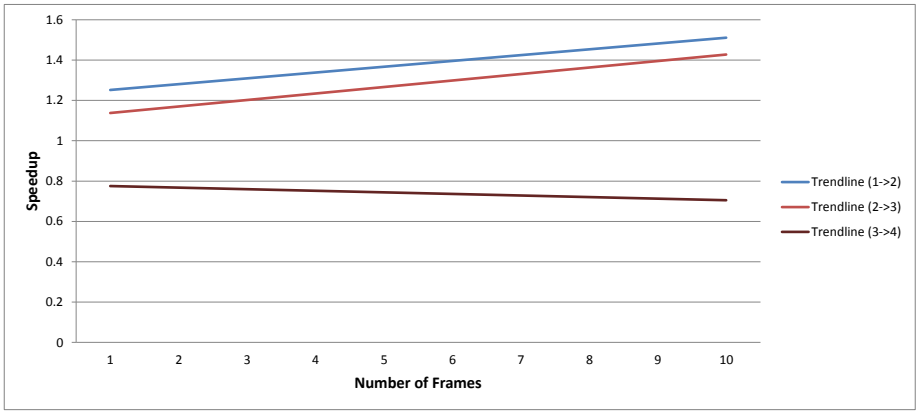


Figure 8.10: Speedup trends when increasing the number of processors for varying number of frames

the second frame, the Δ-time difference starts reducing, and at the 10^{th} frame, it reduces to 1%.

If we increase the number of frames, we observe that the Δ-times for the calculated results remain almost the same. But for the measured results, the Δ-times reduce with an increase in the number of frames. The reason probably is that in the measured results, the Δ-times depend on the pipeline depth. If we have higher number of frames in the pipeline, the hardware platform experiences fewer context switches and therefore requires shorter time to produce the next frame.

Thus, we can conclude that if we increase the number of frames from 10, the pipeline depth increases resulting in faster production of frames by the hardware platform. As a result, the Δ-times will reduce further, which will also reduce the overall difference between the calculated and measured results.

8.5.2 Speedup Evaluation

Here, we analyse the speedup of the face recognition application system based on the measured results. Figure 8.10 shows the speedup trends achieved by increasing the number of processors for varying number of frames. The highest speedup is achieved when the number of processors is increased from one to two, followed by the increase in the number of processors from two to three. As discussed earlier, due to interference from operating system, the speedup gets worse when the number of processors is increased from three to four.

8.5.3 Tool Evaluation

This section reports on computation times and memory consumption of our approach when applied in the case study of face recognition system application. Recall from Chapter 4 that we use the *fastest* trace option in UPPAAL to achieve

Number of Frames	Computation Time (Number of Processors)				Memory Consumption (Number of Processors)			
	1	2	3	4	1	2	3	4
1	26.5	152.05	69.95	175.83	1095.1	23071.376	18623.908	36474.805
2	-	-	-	-	-	-	-	-

Table 8.11: Computation time (min) and memory consumption (MB) when using breadth first search. "-" denotes that the experiment is aborted if the run time has exceeded 8 hours.

maximum throughput. Furthermore, the *breadth first* search order is used to search the state-space. Breadth first search is typically the most efficient option when the complete state-space must be searched which ensures optimality.

When using the *breadth first* search order, Table 8.11 shows the computation times (min) and the memory consumption (MB) of our approach. The experiments are run on a high-performance cluster server with 72 cores and 74 GB memory. Note that "-" denotes that the experiment is aborted if the run time has exceeded 8 hours. We can see that the *breadth first* search order is able to produce results for one frame only within 8 hours.

Alternatively, we have used the *random optimal depth first* (abbreviated RODF) search order to obtain the results reported earlier in Table 8.6. The RODF search order randomly searches the successor states and generates a time-optimal trace out of all witnessed traces. Furthermore, traces may vary from run to run.

Using the RODF search order, we run an experiment for some period of time, and then terminate it and observe the completion time of the trace. In this way, we run the same experiment multiple times and select the trace which has the shortest completion time. Even though we cannot guarantee optimality using the RODF search order, we can, at least, get a trace using which we have evaluated the performance of our approach in an industrial case study.

8.6 Conclusions

In this chapter, we have validated the performance of our approach presented in Chapter 4 in a realistic scenario. The case study considered in this chapter is of face recognition system. The experimental setup has been also explained in detail before presenting the results. The results also have helped to understand the following trends of the case study.

- Difference between the calculated and measured results which is 28% at the final frame.
- Difference between the calculated and measured results for initial frames

shows a transient behaviour followed by a steady-state behaviour for later frames.

- Δ-time difference between the calculated and measured results which is 10% for processing 10 frames.
- Scalability evaluation of our approach shows that random optimal depth first search is more feasible than breadth first search which breaks down completely for large scale models.

Currently, the SDF graph is extracted from runs on a single processor. As a future work, we plan to extract SDF graphs from multiprocessor runs. This will give a better indication of the performance of our approach.

Acknowledgements. The author thanks Recore Systems for providing us with the face recognition system case study and validation of the generated schedules, in particular, Kim Sunesen and Timon ter Braak for fruitful discussions about the case study and help with analysis.

Conclusions

APPLICATIONS for embedded systems are continuously widening, e.g., 3D-enabled mobile phones, virtual reality gaming consoles, self-driving cars, nano-satellites etc. On the one hand, this trend is improving the standards of a human life. On the other hand, this trend poses challenges such as increase in energy consumption leading to depletion of world's energy sources. This thesis presents several performance and energy optimisation methodologies, as well as smart designing of these applications. We consider SDF graphs as a model of computation to model streaming applications because SDF graphs naturally capture the characteristics of streaming applications and allow design-time analysis of timing and resource usage. Furthermore, we consider (homogeneous/heterogeneous) hardware platforms onto which streaming applications are mapped. This chapter provides an overview of the research results achieved in this thesis, and gives recommendations for future work.

9.1 Contributions

This thesis provides novel methods for performance and energy optimisation of streaming applications. The research results presented in this thesis contribute to the following main research question.

'*How to manage performance and energy of streaming applications running on a given number of (possibly heterogeneous) processors with respect to their hard real-time requirements.*'

Performance management. The first part of the research question, i.e., performance management, is approached by formalising software components (SDF actors, channels) that form a (software) streaming application and hardware components (processors) that together form a platform application model (PAM) in Chapter 4. We also have given definitions of resource capacities (number of processors) and performance (throughput). To achieve the maximum throughput of an SDF graph on a limited number of processors, the SDF graph and PAM are translated to the UPPAAL model checker automatically using the STARS tool-chain. Afterwards, using the *fastest trace* generation option of UPPAAL, the throughput-optimal trace is generated which can be visualised as a schedule using SDF FISH.

Energy management. The second part of the research question i.e., energy management, is addressed by extending the definition of PAMs with the power consumption values and clustering processors into VFIs in Chapter 5. We have used the following state-of-the-art energy reduction techniques.

- Dynamic Voltage and Frequency Scaling (DVFS) to lower the dynamic power consumption, and
- Dynamic Power Management to reduce the static power consumption.

In this work, an SDF graph and a PAM are translated to the UPPAAL CORA model checker in an automated fashion using the COMET tool-chain. The energy optimisation problem is encoded as a query over priced timed automata. As a result, UPPAAL CORA computes energy-optimal schedules.

Analysis of battery-powered systems. Energy management of streaming applications is taken one step forward by including the battery system consisting of a limited number of batteries and battery capacities in Chapter 6. The whole system, including an SDF graph, a PAM, and multiple kinetic battery models (KiBaMs) are modelled as hybrid automata. Then, using the statistical model checker UPPAAL SMC, Monte Carlo simulations are applied to evaluate (1) system lifetime; and (2) minimum required initial battery capacities to achieve the desired throughput.

These contributions are the product of several other contributions of this thesis, both theoretical and practical, as given below.

More expressive models. Most of earlier works on energy reduction and analysis of battery-powered systems either consider directed acyclic graphs (DAGs) without analysing periodicity; or frame-based periodic applications with no data dependencies between periods. In contrast, we consider SDF graphs as a MoC for streaming applications which are more expressive and better represent streaming applications.

More powerful methodologies. This thesis contributes to the field of scheduling and analysis of streaming applications by providing powerful methodologies, as described in the following.

- Existing techniques of generating schedules for SDF graphs on multiprocessors consider either transforming SDF graphs to other forms such as HSDF graphs and directed acyclic graphs (DAGs), or considering self-timed execution which assumes to have sufficient processors to accommodate all the enabled executions simultaneously. As opposed to these techniques, this thesis provides a methodology of generating throughput-optimal schedules on a given number of processors without transforming the SDF graphs.
- Existing techniques use specific scheduling schemes such as round-robin or TDM which cannot guarantee optimality. This thesis considers non-deterministic scheduling, and whole state-space is explored to search for the optimal schedules.

- Contemporary tools such as SDF^3 do not support verification of functional system correctness or detection of subtle design errors in early phases. This thesis addresses this lack by bridging the gap between SDF analysis and model checking. This allows verification of (preservation of) user-defined properties such as the absence of deadlocks, safety, and liveness.

Maintainable toolset. We have developed a model-driven framework for HW-SW co-design of SDF graphs mapped on a hardware platform. In this framework, a reusable set of three coherent metamodels is proposed. To provide interoperability among SDF and model-checking domains, a reusable set of extensible model transformations is defined. The metamodel developed for SDF graphs can be utilised by SDF community to transform SDF models to other domains and vice versa.

Practical validation. Our experimental evaluation of performance analysis in an industrial case study confirms the validity and effectiveness of our method. This contributes to the field of model checking by demonstrating its application in large scale case studies.

9.2 Recommendations for Future Work

As proposed in Chapter 1, the approach presented in this thesis consists of four ingredients, namely (1) a model of computation for streaming applications (SDF graphs), (2) a hardware platform model, (3) an analysis environment (model checking), and (4) a modelling environment (model-driven engineering). For each of these ingredients, we propose the following recommendations for future work.

1. Model of computation for streaming applications. Although SDF graphs offer fast analysis algorithms, and allow efficient implementation, they are not very expressive. As a future work, we envision to consider more expressive models of computation such as Scenario-Aware Dataflow (SADF) or multidimensional synchronous dataflow (MDDF) [Lee93, ML02], which generalises scalar consumption and production rates to tuples that specify multidimensional index spaces.

2. Hardware platform model.

Shared Resources. This thesis does not consider any interprocessor communication and shared resources. This can be achieved by extending PAMs with an interconnect and a shared FIFO buffer. The interconnect will arbitrate requests from different processors according to some arbitration mechanisms such as First-Come-First-Serve, Round-Robin, and Fixed-priority, and transport tokens to the shared FIFO buffer.

Changing frequency level during actor firing. Currently in the energy-optimal scheduling approach, once an actor starts firing at a certain frequency, the processor cannot change the frequency level until the end of the firing. An advanced energy optimisation methodology which considers this possibility can be modelled using stopwatch automata [CL00] (timed automata with stopwatches) in order to record the elapsed execution times of the actors being fired.

3. Analysis environment. This thesis has discussed energy-optimal scheduling of battery-powered systems. However, battery-aware scheduling has not been considered. One of the reasons for this was the lack of suitable tools which can model hybrid aspects of KiBaMs, and nondeterminism together. This limitation can be overcome by modelling in stochastic hybrid games [LMM$^+$16] which combine priced timed automata, hybrid automata, and timed games [CDF$^+$05]. Stochastic hybrid games can be analysed using the UPPAAL STRATEGO tool [DJL$^+$15] which provides synthesis of safe and near-optimal strategies for stochastic hybrid games, using a combination of symbolic synthesis, statistical model checking, and reinforcement learning. A first sketch of this approach is published in [AvdP16]. In this work, energy-optimal solutions are derived using UPPAAL STRATEGO by first synthesising a permissive controller satisfying a throughput constraint, and then select a near-optimal strategy that additionally minimises the energy consumption.

We believe that battery-aware energy-optimal scheduling can be achieved by following the same line of research, as stochastic hybrid games allow modelling of both hybrid aspects of KiBaMs, and nondeterminism together.

4. Modelling environment. Currently, our model-driven framework termed COMET includes only one tool for the analysis of SDF graphs, i.e., SDF^3. For considering more expressive models of computation for streaming applications such as MDDF graphs, we must also consider to include relevant tool support in the COMET framework.

As a future work, we plan to provide interoperability between UPPAAL and a state-of-the-art tool for modelling and analysis of MDDF graphs termed Ptolemy [BHLM94]. This can be done by extending the framework with the metamodels for MDDF graphs and Ptolemy, and developing model transformation from Ptolemy to UPPAAL.

With every passing year, (non-renewable) energy sources are getting scarce. Therefore it is a joint responsibility of civil society, government, and industry to pave a path towards green computing in order to secure a better future for our coming generations. The author hopes that this thesis provides a crucial stepping stone along this path.

Part IV

Appendices

Detailed Translation of System Model to Hybrid Automata (Chapter 6)

Here, we give the detailed translation of the system model defined in Chapter 6 to hybrid automata. Let us consider the example of an MPEG-4 decoder in Figure 6.6 on page 135 mapped on Samsung Exynos 4210 processors which are powered by three batteries $B = \{bat_i, bat_k, bat_l\}$ such that $k = i + 1$ and $l = k + 1$.

Hybrid Automaton $\mathcal{K}_{sched}$. The hybrid automaton $\mathcal{K}_{sched}$ models the scheduling scheme of batteries, i.e., round-robin. After an iteration is finished, this automaton switches from the battery $bat_i \in B$ to the next available battery $bat_k \in B$. If the battery $bat_k \in B$ has already run out of charge, $\mathcal{K}_{sched}$ will switch to the next battery, i.e., $bat_l \in B$, and so on. Figure 1.1 shows the automaton $\mathcal{K}_{sched}$, for three batteries.

The automaton $\mathcal{K}_{sched}$ is defined as,

$$\mathcal{K}_{sched} = (L, l^0, Act, X, E, F, Inv)$$

We define all components of the $\mathcal{K}_{sched}$ as follows.

- For each battery $bat_y \in B$, we include a location $L = \{\mathsf{avail\_baty}\}$ to indicate which the battery is currently active.
- For $B = \{bat_1, bat_2, \ldots, bat_m\}$, the initial location is, $l^0 = \mathsf{avail\_bat1}$, indicating that the battery bat_1 serves first.
- The hybrid automaton $\mathcal{K}_{sched}$ has an action $Act = \{\mathsf{startNextIter?}\}$ to synchronise with G_{obs} when the current iteration finishes, so that $\mathcal{K}_{sched}$ can choose the next battery for the next iteration.
- The hybrid automaton $\mathcal{K}_{sched}$ does not contain any continuous variable.
- The hybrid automaton $\mathcal{K}_{sched}$ has a variable: $\mathsf{active\_KiBaM\_id}$ that determines the currently active battery. For $B = \{bat_1, bat_2, \ldots, bat_m\}$, the initial value of $\mathsf{active\_KiBaM\_id}$=1, indicating that the battery bat_1 is the first to serve. For each battery $bat_i \in B$, $bat_k \in B$, and $bat_l \in B$ (for $k = i + 1$ and $l = k + 1$), the edge set E have the following edges.
  - $\mathsf{avail\_bati} \xrightarrow{\mathsf{on\_k!=0:\ startNextIter?,\ active\_KiBaM\_id=k}} \mathsf{avail\_batk}$

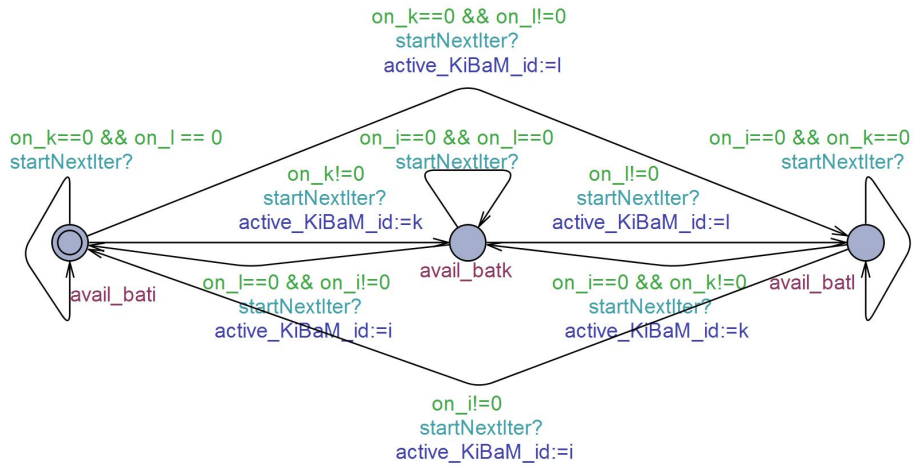


Figure 1.1: $\mathcal{K}_{sched}$ modelling battery scheduler

- avail_bati $\xrightarrow{\text{on_k=0∧on_l!=0: startNextIter?, active_KiBaM_id=l}}$ avail_batl
- avail_bati $\xrightarrow{\text{on_k=0∧on_l=0: startNextIter?, ∅}}$ avail_bati

  ⋮
- avail_batl $\xrightarrow{\text{on_i!=0: startNextIter?, active_KiBaM_id=i}}$ avail_bati
- avail_batl $\xrightarrow{\text{on_i=0∧on_k!=0: startNextIter?, active_KiBaM_id=k}}$ avail_batk
- avail_batl $\xrightarrow{\text{on_i=0∧on_k=0: startNextIter?, ∅}}$ avail_batl

  After an iteration finishes, the action startNextIter synchronises with G_{obs} to start a new iteration. But, before the new iteration starts, the next available battery is selected, and all other batteries are going to stay idle in the meanwhile.

- There are no delay functions in $\mathcal{K}_{sched}$.
- We do not have any clocks and invariants in $\mathcal{K}_{sched}$. Therefore, $Inv(l) = true$ for all $l \in L$.

Hybrid Automata $\mathcal{K}_y$. The HA $\mathcal{K}_1, \ldots, \mathcal{K}_m$ model the batteries $B = \{bat_1, \ldots, bat_m\}$. The model of $bat_y \in B$ is shown in Figure 1.2. The HA $\mathcal{K}_y$ inform $\mathcal{K}_{obs}$, when the battery bat_y gets empty.

For each $bat_y \in B$, the HA $\mathcal{K}_y$ is defined as,

$$\mathcal{K}_y = (L_y, l_y^0, Act_y, X_y, E_y, F_y, Inv_y)$$

All components of $\mathcal{K}_y$ are given in the following.

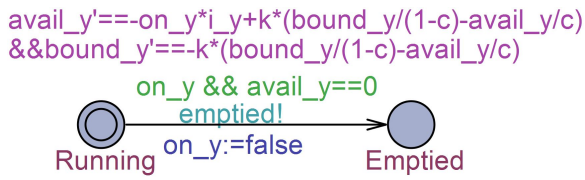


Figure 1.2: $\mathcal{K}_y$ modelling bat_y

- The HA $\mathcal{K}_y$ contain two locations $L_y = \{\mathsf{Running}, \mathsf{Emptied}\}$ to denote if a battery is running or emptied.

- The initial location is given by $l_y^0 = \{\mathsf{Running}\}$. This explains that a battery $b_y \in B$ always starts in the running state.

- There is an action in $\mathcal{K}_y$, i.e., $Act_y = \{\mathsf{emptied!}\}$ to synchronise with $\mathcal{K}_{obs}$.

- The HA $\mathcal{K}_y$ contain two continuous variables $X_y = \{\mathsf{avail\_y}, \mathsf{bound\_y}\}$ to denote the available and bound charge in $bat_y \in B$, respectively.

- The HA $\mathcal{K}_y$ contain a number of variables: a boolean variable $\mathsf{on\_y}$ to determine if the battery has available charge or whether it has run out of it; and a variable $\mathsf{i\_y}$ to annotate the load current being consumed from $bat_y \in B$. Initially, we have $\mathsf{on\_y} = \mathit{true}$ and $\mathsf{i\_y} = 0$. The edge set E_y has only edge, given as follows.

  – $\mathsf{Running} \xrightarrow{\mathsf{on\_y} \wedge \mathsf{avail\_y}=0\text{: emptied!, on\_y}=\mathit{false}} \mathsf{Emptied}$

  The above edge synchronises with $\mathcal{K}_{obs}$ over the urgent action $\mathsf{emptied!}$, and is taken if the available charge $\mathsf{avail\_y}$ reaches zero, emphasising that the battery $bat_y \in B$ is empty. As a result of this action, the value of $\mathsf{on\_y}$ changes to false.

- The initial location l_y^0 uses equations (6.2) and (6.3) as a delay function. For convenience, we give equations (6.2) and (6.3) in the following.

$$\dot{a}(t) = -i(t) + k(h_b - h_a)$$

$$\dot{b}(t) = -k(h_b - h_a)$$

  This represents that, as long as $bat_y \in B$ is non-empty, the available and bound charge of $\mathcal{K}_y$ evolves according to the equations (6.2) and (6.3) respectively.

- We do not have any clocks and invariants in $\mathcal{K}_y$. Therefore, $Inv(l) = \mathit{true}$ for all $l \in L$.

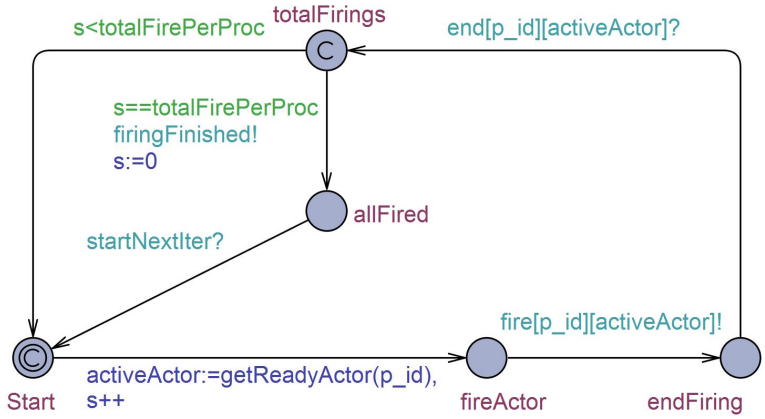


Figure 1.3: G_sched_j modelling scheduler for processor π_j

Hybrid Automata G_sched_j. The HA G_sched_j implement static-order (SO) firings of SDF actors on processors. For this purpose, after G_{obs} informs G_sched_j that an iteration has started, G_sched_j map actors on $Processor_j$ according to the SO schedule of that processor. When all actors are fired according to the SO schedule of $Processor_j$, G_sched_j inform G_{obs} back, indicating completion of the current iteration. For a processor $\pi_j \in \Pi$, Figure 1.3 presents the HA G_sched_j.

For each $\pi_j \in \Pi$, G_sched_j is defined as,

$$G\_sched_j = (L_j, l_j^0, Act_j, X_j, E_j, F_j, Inv_j)$$

where

- The location set contains five locations $L_j = \{\mathsf{Start}, \mathsf{fireActor}, \mathsf{endFiring}, \mathsf{totalFirings}, \mathsf{allFired}\}$.
- The initial location is given by $l_j^0 = \{\mathsf{Start}\}$.
- The HA G_sched_j contain four actions, i.e., $Act_j = \{\mathsf{fire!}, \mathsf{end?}, \mathsf{startNextIter?}, \mathsf{firingFinished!}\}$. The actions fire and end are parametrised with processor and action ids, and are used to synchronise with $Processor_j$. The actions startNextIter and firingFinished synchronise G_sched_j with G_{obs}.
- There are no continuous variables in G_sched_j.
- The HA G_sched_j have a number of local variables: activeActor_j that determines the active actor currently mapped on the processor π_j; and s_j that determines the index of the active actor in the SO schedule. Initially, activeActor_j $= 0$, and s_j $= 0$. The HA G_sched_j also contain a parametrised variable totalFirePerProc_j, that defines the total number of tasks in the SO schedule of the processor π_j. Since these variables are local,

we can abbreviate them by activeActor, s and totalFirePerProc respectively. The edge set E_j is explained below.

- The following edge fetches the active actor (on a specific processor) according to the SO schedule for each processor π_j, using the function getReadyActor(j). As a result of this edge, the value of s is incremented by one, which means that the next actor in the SO schedule is fetched next time.

  Start $\xrightarrow{true:\ \varnothing,\ \text{activeActor=getReadyActor(j)}\wedge\text{s++}}$ fireActor

- The following edge maps the fetched (active) actor, on the processor automaton $Processor_j$, using the action fire!.

  fireActor $\xrightarrow{true:\ \text{fire[j][activeActor]!},\ \varnothing}$ endFiring

- In the following edge, the urgent action end? synchronises with the processor automaton $Processor_j$. As a result, $Processor_j$ informs G_sched_j that the firing of the active actor has finished.

  endFiring $\xrightarrow{true:\ \text{end[j][activeActor]?},\ \varnothing}$ totalFirings

- The following edge checks if the SO schedule of a processor π_j is not fully executed, using the guard condition s < totalFirePerProc. If this is the case, the following edge is taken, leading to the Start location where the next actor in the SO schedule is fetched.

  totalFirings $\xrightarrow{\text{s<totalFirePerProc:}\ \varnothing,\ \varnothing}$ Start

- If all actors in the SO schedule of a processor π_j are fired as checked by the guard condition s == totalFirePerProc on the following edge, the action firingFinished! synchronises with the observer automaton G_{obs}. In this way, G_sched_j informs G_{obs} that the processor π_j has fired all mapped actors in the current iteration. The variable s is also reset.

  totalFirings $\xrightarrow{\text{s=totalFirePerProc: firingFinished!, s=0}}$ allFired

- The following edge synchronises with the observer automaton G_{obs} on the action startNextIter? to start executing the SO schedule of the next iteration.

  allFired $\xrightarrow{true:\ \text{startNextIter?},\ \varnothing}$ Start

- There are no delay functions in G_sched_j.
- The HA G_sched_j do not contain any invariants and clocks. Thus, $Inv(l) = true$ for all $l \in L$.

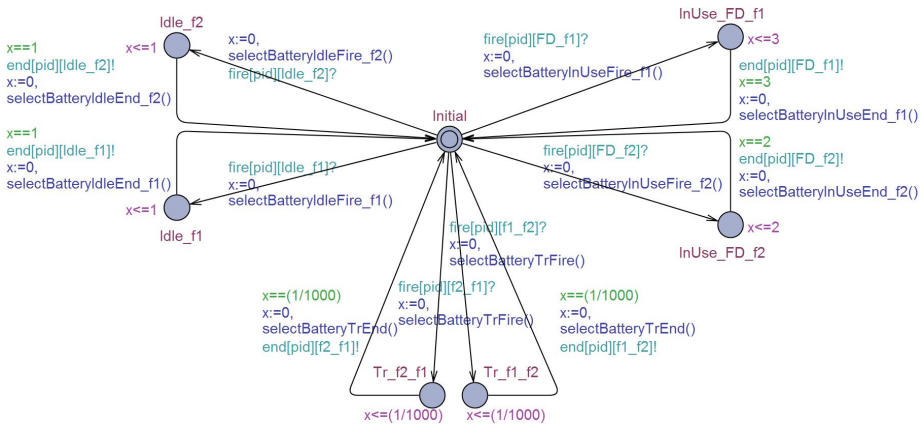


Figure 1.4: $Processor_j$ showing processor model with respect to actor FD

Hybrid Automata $Processor_j$. Likewise, the HA $Processor_1, \ldots, Processor_n$ model the processors $\Pi = \{\pi_1, \ldots, \pi_n\}$, as shown in Figure 1.4. For better visibility, Figure 1.4 shows the HA of $Processor_j$, with respect to one actor only, i.e., $FD \in A$. The actors in the SO schedule of a processor π_j are mapped on the HA $Processor_j$ by the HA G_sched_j, using the actions fire and end. Please note that $Processor_j$ contains exactly one clock x_j; since clocks in UPPAAL SMC are local, we can abbreviate x_j by x. A separate clock variable global observes the overall time progress.

For each $\pi_j \in \Pi$, we define HA as,

$$Processor_j = (L_j, l_j^0, Act_j, X_j, E_j, F_j, Inv_j)$$

We define all components of $Processor_j$ in the following.

- For each frequency level $f_i \in F$, we include both an idle state and an active state running on that frequency level. Thus, for each $a \in \zeta(\pi_j)$ and $F = \{f_1, \ldots, f_m\}$ such that $f_1 < f_2 < \ldots < f_m$, let $L_{mapping} = \{\mathsf{Idle\_f1}, \ldots, \mathsf{Idle\_fm}, \mathsf{InUse\_a\_f1}, \ldots, \mathsf{InUse\_a\_fm}\}$ indicating that the processor $\pi_j \in \Pi$ is currently used by the actor $a \in A$ at the frequency level $f_i \in F$, either in idle or running state. Furthermore, for $F = \{f_1, \ldots, f_m\}$ such that $f_1 < f_2 < \ldots < f_l < f_m$, we include locations which define overhead of switching between frequency levels, such that $L_{overhead} = \{\mathsf{Tr\_f1\_f2}, \mathsf{Tr\_f2\_f1}, \ldots, \mathsf{Tr\_fl\_fm}, \mathsf{Tr\_fm\_fl}\}$. Moreover, we also have $L_{init} = \{\mathsf{Initial}\}$. Thus, $L_j = L_{mapping} \cup L_{overhead} \cup L_{init}$.
- The initial location is defined as $l_j^0 = L_{init} = \{\mathsf{Initial}\}$.
- The action set $Act_j = \{\mathsf{fire?}, \mathsf{end!}\}$ contains two broadcast actions fire?, end!. The actions fire? and end! in Act_j are parametrised with processor and actor ids with frequencies, and synchronise with G_{sched}.

Listing A.1: selectBatteryInUseFire_fi() Function

```
1 double selectBatteryInUseFire_fi()
2 {
3   if (active_KiBaM_id == k)
4   {
5   return i_k = i_k + I_occ(π_j, f_i);
6   }
7   else
8   return i_l = i_l + I_occ(π_j, f_i);
9 }
```

Listing A.2: selectBatteryInUseEnd_fi() Function

```
1 double selectBatteryInUseEnd_fi()
2 {
3   if (active_KiBaM_id == k)
4   {
5   return i_k = i_k − I_occ(π_j, f_i);
6   }
7   else
8   return i_l = i_l − I_occ(π_j, f_i);
9 }
```

- We do not have any continuous variables in $Processor_j$.
- For each $\pi \in \Pi$, $a \in \zeta(\pi)$ and $f_i \in F$, the edge set E_j contains two edges such that:
  - $\mathsf{Initial} \xrightarrow{true:\ \mathsf{fire}[\pi][\mathsf{a\_fi}]?,\ \mathsf{x=0} \wedge \mathsf{selectBatteryInUseFire\_fi}()} \mathsf{InUse\_a\_fi}$, and
  - $\mathsf{InUse\_a\_fi} \xrightarrow{x=\tau_{act}(a,f_i):\ \mathsf{end}[\pi][\mathsf{a}]!,\ \mathsf{selectBatteryInUseEnd\_fi}()} \mathsf{Initial}$.

  For $bat_k \in B$ and $bat_l \in B$, $\pi_j \in \Pi$ and $f_i \in F$, the functions selectBatteryInUseFire_fi() and selectBatteryInUseEnd_fi() are defined in Listings A.1 and A.2 respectively.

  The action fire$[\pi]$[a_fi] is enabled in the initial state Initial and leads to the location InUse_a_fi. Thus, the action fire$[\pi]$[a_fi] is taken, if the actor $a \in A$ is supposed to claim the processor $\pi \in \Pi$ at the frequency level $f_i \in F$ in the SO schedule. As each location InUse_a_fi has an invariant $Inv_j(\mathsf{InUse\_a\_fi}) \leq \tau_{act}(\mathsf{a}, \mathsf{f_i})$, the automaton can stay in InUse_a_fi for at most the execution time of actor $a \in A$ at the frequency level $f_i \in F$, i.e., $\tau_{act}(a, f_i)$. If $x = \tau_{act}(a, f_i)$, the system has to leave the location InUse_a_fi at exactly the execution time of actor $a \in A$ at the frequency level $f_i \in F$, by taking the end$[\pi]$[a_fi] action.

Listing A.3: selectBatteryIdleFire_fi() Function

```
1 double selectBatteryIdleFire_fi()
2 {
3   if (active_KiBaM_id == k)
4   {
5   return i_k = i_k + I_idle(π_j, f_i);
6   }
7   else
8   return i_l = i_l + I_idle(π_j, f_i);
9 }
```

Listing A.4: selectBatteryIdleEnd_fi() Function

```
1 double selectBatteryInUseEnd_fi()
2 {
3   if (active_KiBaM_id == k)
4   {
5   return i_k = i_k − I_idle(π_j, f_i);
6   }
7   else
8   return i_l = i_l − I_idle(π_j, f_i);
9 }
```

For each $\pi \in \Pi$, and $f_i \in F$, the edge set E_j contains two more edges such that:

- Initial $\xrightarrow{true:\ \mathsf{fire}[\pi][\mathsf{idle\_fi}]?,\ \mathsf{x}=0 \wedge \mathsf{selectBatteryIdleFire\_fi}()}$ Idle_fi, and
- Idle_fi $\xrightarrow{x=1:\ \mathsf{end}[\pi][\mathsf{idle\_fi}]!,\ \mathsf{selectBatteryIdleEnd\_fi}()}$ Initial.

For $bat_k \in B$ and $bat_l \in B$, $\pi_j \in \Pi$ and $f_i \in F$, the functions selectBatteryIdleFire_fi() and selectBatteryIdleEnd_fi() are defined in Listings A.3 and A.4 respectively.

The action fire$[\pi]$[idle_fi] is enabled in the initial location Initial and leads to the location Idle_fi. Thus, fire$[\pi]$[idle_fi] causes the processor $\pi \in \Pi$ to go to Idle_fi at the frequency level $f_i \in F$, whenever the processor $\pi \in \Pi$ is supposed to be idle at the frequency level $f_i \in F$ in the SO schedule. A processor stays in the occupied state only for the time period, when an actor is mapped on it. However, the idle time spent by a processor $\pi_j \in \Pi$ is not a fixed time interval, and a processor $\pi_j \in \Pi$ can stay idle for any finite period of time. Thus, for convenience, we divide the idle time interval in the SO schedule into time slots of one time unit. Furthermore, each location Idle_fi has an invariant $Inv_j(\mathsf{Idle\_fi}) \leq 1$, which means that the

automaton can stay in Idle_fi for at most 1 time unit. If $x = 1$, the system has to leave Idle_fi at exactly one time unit, by taking the end$[\pi]$[idle_fi] action.

For $F = \{f_1, \ldots, f_l, f_m\}$ such that $f_1 < f_2 < \ldots < f_l < f_m$, and $\pi_j \in \Pi$, the edge set E_j has the following edges also.

- Initial $\xrightarrow{true:\ \mathsf{fire}[\pi][\mathsf{f1\_f2}]?,\ \mathsf{x{=}0}\wedge\mathsf{selectBatteryTrFire}()}$ Tr_f1_f2,
- Tr_f1_f2 $\xrightarrow{x=T_{tr}(\pi,f_1,f_2):\ \mathsf{end}[\pi][\mathsf{f1\_f2}]!,\ \mathsf{selectBatteryTrEnd}()}$ Initial,
- Initial $\xrightarrow{true:\ \mathsf{fire}[\pi][\mathsf{f2\_f1}]?,\ \mathsf{x{=}0}\wedge\mathsf{selectBatteryTrFire}()}$ Tr_f2_f1,
- Tr_f2_f1 $\xrightarrow{x=T_{tr}(\pi,f_2,f_1):\ \mathsf{end}[\pi][\mathsf{f2\_f1}]!,\ \mathsf{selectBatteryTrEnd}()}$ Initial,
- $\vdots$
- Initial $\xrightarrow{true:\ \mathsf{fire}[\pi][\mathsf{fl\_fm}]?,\ \mathsf{x{=}0}\wedge\mathsf{selectBatteryTrFire}()}$ Tr_fl_fm,
- Tr_fl_fm $\xrightarrow{x=T_{tr}(\pi,f_l,f_m):\ \mathsf{end}[\pi][\mathsf{fl\_fm}]!,\ \mathsf{selectBatteryTrEnd}()}$ Initial,
- Initial $\xrightarrow{true:\ \mathsf{fire}[\pi][\mathsf{fm\_fl}]?,\ \mathsf{x{=}0}\wedge\mathsf{selectBatteryTrFire}()}$ Tr_fm_fl,
- Tr_fm_fl $\xrightarrow{x=T_{tr}(\pi,f_m,f_l):\ \mathsf{end}[\pi][\mathsf{fm\_fl}]!,\ \mathsf{selectBatteryTrEnd}()}$ Initial

The action fire$[\pi]$[fl_fm] causes the processor $\pi \in \Pi$ to incur the transition overhead, whenever the processor $\pi \in \Pi$ is supposed to change the frequency $f_l \in F$ to $f_m \in F$ in the SO schedule, and so on. The transition overhead is incurred using the functions selectBatteryTrFire() and selectBatteryTrEnd().

- The HA $Processor_j$ do not contain any delay functions.
- For each location InUse_a_fi $\in L_j$, we have an invariant $Inv_j(\mathsf{InUse\_a\_fi}) \leq \tau_{act}(a, f_i)$ enforcing the system to stay in InUse_a_fi for at most the execution time $\tau_{act}(a, f_i)$. As mentioned earlier, we divide the idle time spent by a processor $\pi_j \in \Pi$ into slots of one time unit, by annotating $Inv_j(\mathsf{Idle\_fi}) \leq 1$. For $F = \{f_1, f_2, \ldots, f_m\}$ such that $f_1 < f_2 < \ldots < f_m$, and $\pi \in \Pi$, $Inv_j(\mathsf{Tr\_f2\_f1}) \leq T_{tr}(\pi, f_1, f_2)$.

Hybrid Automaton G_{obs}. The SDF graph observer automaton G_{obs} observes if each processor has fired its all mapped actors according to its SO schedule. The automaton G_{obs} also counts the number of finished iterations. Figure 1.5 on the following page shows the hybrid automaton model of G_{obs}.

The automaton G_{obs} is defined as,

$$G_{obs} = (L, l^0, Act, X, E, F, Inv)$$

where

- The location set is defined as $L = \{\mathsf{Initial}, \mathsf{Off}\}$.
- The initial location is defined as $l^0 = \{\mathsf{Initial}\}$.
- The set of actions is defined as, $Act = \{\mathsf{firingFinished?}, \mathsf{startNextIter!}, \mathsf{allEmptied?}\}$.

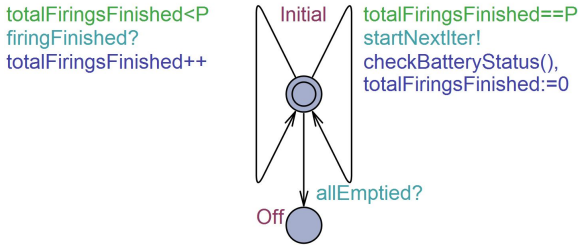


Figure 1.5: G_{obs} modelling SDF observer

- The automaton G_{obs} does not contain continuous variables.
- The automaton G_{obs} has a number of variables: an integer variable P to determine the total number of processors, i.e., $\mathsf{P} = n(\Pi)$; an integer variable Tot_Iter to count the number of finished iterations; and an integer variable totalFiringsFinished to count the number of finished firings in an iteration. Initially, $\mathsf{Tot\_Iter} = 0$ and $\mathsf{totalFiringsFinished} = 0$. The edge set E is described below.
  - In the following edge, the guard condition $\mathsf{totalFiringsFinished} < \mathsf{P}$ checks if less than P number of processors have finished the SO mappings assigned to them. If this is the case, the following edge is synchronised with G_sched_j over the action firingFinished?. As a result, totalFiringsFinished is incremented by one.

    $$\mathsf{Initial} \xrightarrow{\mathsf{totalFiringsFinished<P:\ firingFinished?,\ totalFiringsFinished++}} \mathsf{Initial}$$

  - If P number of processors have executed all mappings assigned to them in an iteration, the following edge is taken. This means that all processors $\pi_j \in \Pi$ are done with executing the SO mappings assigned to them, and an iteration is finished. The automaton G_{obs} also informs all instances of the automaton G_sched_j to start next iteration, by synchronising over the action startNextIter. The function checkBatteryStatus() checks whether the active battery has not got emptied during the iteration. If this is the case, the value of variable Tot_Iter is increased by one.

    $$\begin{array}{l}\mathsf{Initial} \\ \mathsf{Initial}\end{array} \xrightarrow{\mathsf{totalFiringsFinished=P:startNextIter!,\ checkBatteryStatus()\wedge totalFiringsFinished=0}}$$

  For $bat_k \in B$ and $bat_l \in B$, the function checkBatteryStatus() is defined in Listing A.5

  - If all batteries are emptied, the automaton $\mathcal{K}_{obs}$ informs G_{obs} via the following edge over the urgent action allEmptied?. This signifies that

Listing A.5: checkBatteryStatus() Function

```
1  void checkBatteryStatus()
2  {
3    if(active_KiBaM_id == k && on_k == true)
4    {
5    Tot_Iter++;
6    }
7    if(active_KiBaM_id == l && on_l == true)
8    {
9    Tot_Iter++;
10   }
11 }
```

the system lifetime has ended, and G_{obs} needs to stop counting the number of finished iterations.

$\mathsf{Initial} \xrightarrow{true:\, \mathsf{allEmptied?},\, \varnothing} \mathsf{Off}$

- There are no delay functions in G_{obs}.
- The HA G_{obs} do not contain any invariants and clocks. Thus, $Inv(l) = true$ for all $l \in L$.

Hybrid Automaton $\mathcal{K}_{obs}$. The KiBaM observer automaton $\mathcal{K}_{obs}$ observes if any battery gets empty. When all batteries get emptied, $\mathcal{K}_{obs}$ synchronises with G_{obs} to inform about the end of the system lifetime. Figure 1.6 shows the hybrid automaton model of $\mathcal{K}_{obs}$.

The automaton $\mathcal{K}_{obs}$ is defined as,

$$\mathcal{K}_{obs} = (L, l^0, Act, X, E, F, Inv)$$

where

- The set of locations and the initial location is defined as, $L = l^0 = \{\mathsf{Initial}\}$.
- The set of actions is defined as, $Act = \{\mathsf{emptied?}, \mathsf{allEmptied!}\}$.
- There are no continuous variables in $\mathcal{K}_{obs}$.
- The automaton $\mathcal{K}_{obs}$ has two variables: an integer variable totBat to determine the total number of batteries in the system, i.e., $\mathsf{totBat} = n(B)$ where $B = \{bat_1, \ldots, bat_m\}$; and an integer variable empty_count to count the number of emptied batteries. Initially, $\mathsf{empty\_count} = 0$. The edge set E is explained as follows.

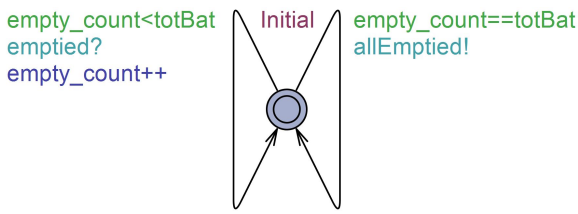


Figure 1.6: $\mathcal{K}_{obs}$ modelling battery observer

– The following edge synchronises with the KiBaM automaton $\mathcal{K}_y$ on the urgent action emptied?, if the battery bat_y is emptied. The guard condition checks if not all batteries are emptied. The variable empty_count is incremented by one as a result of taking this edge.

$$\text{Initial} \xrightarrow{\text{empty\_count<totBat: emptied?, empty\_count++}} \text{Initial}$$

– If all batteries are emptied, the following edge synchronises with G_{obs} to inform about the end of the system lifetime.

$$\text{Initial} \xrightarrow{\text{empty\_count=totBat: allEmptied!, } \varnothing} \text{Initial}$$

- There are no delay functions in $\mathcal{K}_{obs}$.
- $\mathcal{K}_{obs}$ does not contain any clocks or invariants. Thus, $Inv(l) = true$ for all $l \in L$.

After modelling whole system, we run the following query, where bound is the time bound on running the simulation, and Tot_Iter is the variable representing the completed number of iterations. As a result, we get a plot, by which we determine the total number of iterations completed within bound time units. We use the same models and query to determine adequate batteries' capacities.

simulate 1 [<= `bound`]{`Tot_Iter`}

List of papers by the author

Application of Model Checking

[1] W. Ahmad, R. de Groote, P. K. F. Hölzenspies, M. I. A. Stoelinga, and J. C. van de Pol. Resource-constrained optimal scheduling of synchronous dataflow graphs via timed automata. In *Proceedings of 14th International Conference on Application of Concurrency to System Design (ACSD)*, pages 72–81, 2014.

[2] W. Ahmad, P. K. F. Hölzenspies, M. I. A. Stoelinga, and J. C. van de Pol. Green computing: power optimisation of VFI-based real-time multiprocessor dataflow applications. In *Proceedings of 18th Euromicro Conference on Digital Systems Design (DSD)*, pages 271–275, 2015.

[3] W. Ahmad, M. R. Jongerden, M. I. A. Stoelinga, and J. C. van de Pol. Model checking and evaluating QoS of batteries in MPSoC dataflow applications via hybrid automata. In *Proceedings of 16th International Conference on Application of Concurrency to System Design (ACSD)*, pages 114–123, 2016.

[4] W. Ahmad and J. C. van de Pol. Synthesizing energy-optimal controllers for multiprocessor dataflow applications with Uppaal Stratego. In *Proceedings of 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques (ISoLA)*, pages 94–113, 2016.

Application of Model-Driven Engineering

[5] W. Ahmad, B. M. Yildiz, A. Rensink, and M. I. A. Stoelinga. A model-driven framework for hardware-software co-design of dataflow applications. In *Proceedings of 6th International Workshop on Design, Modeling and Evaluation of Cyber Physical Systems (CyPhy)*, 2016.

Technical Reports

[6] W. Ahmad, R. de Groote, P. K. F. Hölzenspies, M. I. A. Stoelinga, and J. C. van de Pol. Resource-constrained optimal scheduling of synchronous dataflow graphs via timed automata. Technical Report TR-CTIT-13-17, Centre for Telematics and Information Technology, University of Twente, 2014.

[7] W. Ahmad, P. K. F. Hölzenspies, M. I. A. Stoelinga, and J. C. van de Pol. Green computing: power optimisation of VFI-based real-time multiprocessor dataflow applications (extended version). Technical Report TR-CTIT-15-04, Centre for Telematics and Information Technology, University of Twente, 2015.

[8] W. Ahmad, M. R. Jongerden, M. I. A. Stoelinga, and J. C. van de Pol. Model checking and evaluating QoS of batteries in MPSoC dataflow applications via hybrid automata (extended version). Technical Report TR-CTIT-16-03, Centre for Telematics and Information Technology, University of Twente, 2016.

[9] W. Ahmad, B. M. Yildiz, A. Rensink, and M. I. A. Stoelinga. A model-driven framework for hardware-software co-design of dataflow applications (extended version). Technical Report CTIT-TR-16-09, Centre for Telematics and Information Technology, University of Twente, 2016.

List of Symbols

s_{nom}	Nominal speed of a task
s_{opt}	Optimal speed of a task
A	Set of actors of an SDF graph
D	Set of dependency channels of an SDF graph
Tok_0	Initial tokens in each channel of an SDF graph
$In(a)$	Input channels of an actor a
$Out(a)$	Output channels of an actor a
$CR(d)$	Consumption rate of a channel d
$PR(d)$	Production rate of a channel d
τ	Execution time of an actor
Tok	Number of tokens in each channel
TuC	Remaining execution time
κ	Type of transition of an SDF graph
χ	Execution of an SDF graph
γ	Repetition vector of an SDF graph
$iter$	Number of iterations per period
Π	Set of processors
π	Processor
ζ	Mapping function from actors to processors
F	Set of frequencies
P_{occ}	Operating power consumption of a processor
P_{idle}	Operating power consumption of a processor
P_{tr}	Transition overhead
τ_{act}	Actual execution time of an actor
$[\pi]$	VFI of a processor π
$status$	Status of a processor
$TotPow$	Accumulated power of a processor
bat	Battery

I_{occ}	Operating load current of a processor
I_{idle}	Idle load current of a processor
I_{tr}	Load current overhead of a transition
T_{tr}	Time overhead of a transition
Δ	Difference in completion times
L	Set of locations
l^0	Initial location
Act	Set of actions
C	Set of clocks
E	Set of edges
Inv	Invariant of each location

References

[AD90] R. Alur and D. L. Dill. Automata for modeling real-time systems. In *Proceedings of 17th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 322–335, 1990.

[AD94] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[AdGH+14a] W. Ahmad, R. de Groote, P. K. F. Hölzenspies, M. I. A. Stoelinga, and J. C. van de Pol. Resource-constrained optimal scheduling of synchronous dataflow graphs via timed automata. In *Proceedings of 14th International Conference on Application of Concurrency to System Design (ACSD)*, pages 72–81, 2014.

[AdGH+14b] W. Ahmad, R. de Groote, P. K. F. Hölzenspies, M. I. A. Stoelinga, and J. C. van de Pol. Resource-constrained optimal scheduling of synchronous dataflow graphs via timed automata. Technical Report TR-CTIT-13-17, Centre for Telematics and Information Technology, University of Twente, 2014.

[AHSvdP15a] W. Ahmad, P. K. F. Hölzenspies, M. I. A. Stoelinga, and J. C. van de Pol. Green computing: Power optimisation of VFI-based real-time multiprocessor dataflow applications. In *Proceedings of 18th Euromicro Conference on Digital System Design (DSD)*, pages 271–275, 2015.

[AHSvdP15b] W. Ahmad, P. K. F. Hölzenspies, M. I. A. Stoelinga, and J. C. van de Pol. Green computing: Power optimisation of VFI-based real-time multiprocessor dataflow applications (extended version). Technical Report TR-CTIT-15-04, Centre for Telematics and Information Technology, University of Twente, 2015.

[AJSvdP16a] W. Ahmad, M. R. Jongerden, M. I. A. Stoelinga, and J. C. van de Pol. Model checking and evaluating QoS of batteries in MPSoC dataflow applications via hybrid automata. In *Proceedings of 16th International Conference on Application of Concurrency to System Design (ACSD)*, 2016.

[AJSvdP16b] W. Ahmad, M. R. Jongerden, M. I. A. Stoelinga, and J. C. van de Pol. Model checking and evaluating QoS of batteries in MPSoC dataflow applications via hybrid automata (extended version). Technical Report TR-CTIT-16-03, Centre for Telematics and Information Technology, University of Twente, 2016.

[AK98] H. A. Andrade and S. Kovner. Software synthesis from dataflow models for G and LabVIEW. In *Proceedings of 32nd Asilomar Conference on Signals, Systems and Computers*, volume 2, pages 1705–1709, 1998.

[AvdP16] W. Ahmad and J. C. van de Pol. Synthesizing energy-optimal controllers for multiprocessor dataflow applications with Uppaal Stratego. In *Proceedings of 7th International Symposium on Leveraging Applications of*

Formal Methods, Verification and Validation: Foundational Techniques (ISoLA), pages 94–113, 2016.

[AYRS16a] W. Ahmad, B. M. Yildiz, A. Rensink, and M. I. A. Stoelinga. A model-driven framework for hardware-software co-design of dataflow applications. In *Proceedings of 6th International Workshop on Design, Modeling and Evaluation of Cyber Physical Systems (CyPhy)*, 2016.

[AYRS16b] W. Ahmad, B. M. Yildiz, A. Rensink, and M. I. A. Stoelinga. A model-driven framework for hardware-software co-design of dataflow applications (extended version). Technical Report TR-CTIT-16-09, Centre for Telematics and Information Technology, University of Twente, 2016.

[BBDM00] L. Benini, A. Bogliolo, and G. De Micheli. A survey of design techniques for system-level dynamic power management. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(3):299–316, 2000.

[BDD05] L. Bondé, C. Dumoulin, and J.-L. Dekeyser. *Advances in Design and Specification Languages for SoCs*, chapter Metamodels and MDA Transformations for Embedded Systems, pages 89–105. Springer Publishing Company, 2005.

[BDL04] G. Behrmann, A. David, and K. G. Larsen. *A Tutorial on Uppaal*, pages 200–236. Springer Berlin Heidelberg, 2004.

[BDL+12] P. E. Bulychev, A. David, K. G. Larsen, M. Mikucionis, D. B. Poulsen, A. Legay, and Z. Wang. UPPAAL-SMC: statistical model checking for priced timed automata. In *Proceedings of 10th Workshop on Quantitative Aspects of Programming Languages and Systems (QAPL)*, pages 1–16, 2012.

[BELP95] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete. Cyclo-static data flow. In *Proceedings of International Conference on on Acoustics, Speech, and Signal Processing (ICASSP)*, volume 5, pages 3255–3258, 1995.

[BELP96] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cycle-static dataflow. *IEEE Transactions on signal processing*, 44(2):397–408, 1996.

[BFH+01a] G. Behrmann, A. Fehnker, T. Hune, K. G. Larsen, P. Pettersson, and J. Romijn. *Efficient Guiding Towards Cost-Optimality in UPPAAL*, pages 174–188. Springer Berlin Heidelberg, 2001.

[BFH+01b] G. Behrmann, A. Fehnker, T. Hune, K. G. Larsen, P. Pettersson, J. Romijn, and F. Vaandrager. *Minimum-Cost Reachability for Priced Time Automata*, pages 147–161. Springer Berlin Heidelberg, 2001.

[BGJ+02] M. Baleani, F. Gennari, Y. Jiang, Y. Patel, R. K. Brayton, and A. Sangiovanni-Vincentelli. HW/SW partitioning and code generation of embedded control applications on a reconfigurable architecture platform. In *Proceedings of 10th International Symposium on Hardware/Software Codesign (CODES)*, pages 151–156, 2002.

[BGK+02] J. Bengtsson, W. O. D. Griffioen, K. J. Kristoffersen, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Automated verification of an audio-control protocol using Uppaal. *The Journal of Logic and Algebraic Programming*, 52:163 – 181, 2002.

[BHLM94] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal of Computer Simulation*, 4:155–182, 1994.

[BHRV13] T. Basten, R. Hamberg, F. Reckers, and J. Verriet. *Model-Based Design of Adaptive Embedded Systems*. Springer Publishing Company, 2013.

[BK08] C. Baier and J.-P. Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[BL91] B. Barrera and E. A. Lee. Multirate signal processing in Comdisco's SPW. In *Proceedings of International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, volume 2, pages 1113–1116, 1991.

[BLR05] G. Behrmann, K. G. Larsen, and J. I. Rasmussen. Optimal scheduling using priced timed automata. *SIGMETRICS Performance Evaluation Review*, 32(4):34–40, 2005.

[BML99] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI signal processing systems*, 21(2):151–166, 1999.

[BSLM96] S. S. Bhattacharyya, S. Shuvra, E. A. Lee, and P. K. Murthy. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, 1996.

[BV00] J. Buck and R. Vaidyanathan. Heterogeneous modeling and simulation of embedded systems in El Greco. In *Proceedings of 8th International Workshop on Hardware/Software Codesign (CODES)*, pages 142–146, 2000.

[CAM13] Camfind. `http://camfindapp.com`, 2013. Accessed: 2016-12-25.

[CC02] P. Chowdhury and C. Chakrabarti. Battery aware task scheduling for a system-on-a-chip using voltage/clock scaling. In *Proceedings of 16th IEEE Workshop on Signal Processing Systems (SIPS)*, pages 201–206, 2002.

[CCE+99] F. Clouté, J.-N. Contensou, D. Esteve, P. Pampagnin, P. Pons, and Y. Favard. Hardware/software co-design of an avionics communication protocol interface system: An industrial case study. In *Proceedings of 7th International Workshop on Hardware/Software Codesign (CODES)*, pages 48–52, 1999.

[CDF+05] F. Cassez, A. David, E. Fleury, K. G. Larsen, and D. Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *Proceedings of 16th International Conference onConcurrency Theory (CONCUR)*, pages 66–80, 2005.

[CE81] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proceedings of 3rd Workshop on Logics of Programs*, pages 52–71, 1981.

[Cha07] B. R. Chalamala. Portable electronics and the widening energy gap. *Proceedings of the IEEE*, 95:2106–2107, 2007.

[CHU08] S. Chu. The energy problem and Lawrence Berkley National Laboratory. `https://www.arb.ca.gov/research/seminars/chu/chu.pdf`, 2008. Accessed: 2016-11-28.

[CIS16] Cisco visual networking index: Global mobile data traffic forecast, 2015-2020. `http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/vni-forecast-qa.pdf`, 2016. Accessed: 2016-08-17.

[CL00] F. Cassez and K. G. Larsen. The impressive power of stopwatches. In *Proceedings of 11th International Conference on Concurrency Theory (CONCUR)*, pages 138–152, 2000.

[DA12] V. Devadas and H. Aydin. On the interplay of voltage/frequency scaling and device power management for frame-based real-time embedded applications. *IEEE Transactions on Computers*, 61(1):31–44, 2012.

[DDL+12] A. David, D. Du, K. G. Larsen, A. Legay, M. Mikučionis, D. Bøgsted Poulsen, and S. Sedwards. Statistical Model Checking for Stochastic Hybrid Systems. *ArXiv e-prints*, 2012.

[DDL+13] A. David, D. Du, K. G. Larsen, A. Legay, and D. Bøgsted Poulsen. Statistical model checking of dynamic networks of stochastic hybrid automata. In *Proceedings of 13th International Workshop on Automated Verification of Critical Systems (AVOCS)*, 2013.

[dG16] R. de Groote. *On the analysis of synchronous dataflow graphs: a system-theoretic perspective*. PhD thesis, University of Twente, 2016.

[DJL+15] A. David, P. G. Jensen, K. G. Larsen, M. Mikučionis, and J. H. Taankvist. Uppaal Stratego. In *Proceedings of 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 206–211, 2015.

[dKBS12] E. de Groote, J. Kuper, H. J. Broersma, and G. J. M. Smit. Max-plus algebraic throughput analysis of synchronous dataflow graphs. In *Proceedings of 38th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pages 29–38, 2012.

[dLJ06] P. de Langen and B. Juurlink. Leakage-aware multiprocessor scheduling for low power. In *Proceedings of 20th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 80–80, 2006.

[DS96] G. De Micheli and M. Sami, editors. *Hardware/Software Co-Design*. Kluwer Academic Publishers, 1996.

[dS15] A. R. da Silva. Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures*, 43:139–155, 2015.

[DSB+11] M. Damavandpeyma, S. Stuijk, T. Basten, M. C. W. Geilen, and H. Corporaal. Hybrid code-data prefetch-aware multiprocessor task graph scheduling. In *Proceedings of 14th Euromicro Conference on Digital System Design (DSD)*, pages 583–590, 2011.

[ECO05] The real energy crisis. `http://www.economist.com/node/3546049`, 2005. Accessed: 2016-08-18.

[ENE16] Enerdata. Total energy consumption. `https://yearbook.enerdata.net`, 2016. Accessed: 2016-11-28.

[EPA13] US environmental protection agency. Fuel economy information. `http://www.fueleconomy.gov/feg/Find.do?action=sbs&id=33367`, 2013. Accessed: 2016-11-17.

[Fak16] M. Fakih. *State-Based Real-Time Analysis of Synchronous Data-flow (SDF) Applications on MPSoCs with Shared Communication Resources*. PhD thesis, University of Oldenburg, 2016.

[FGFR13] M. Fakih, K. Grüttner, M. Fränzle, and A. Rettberg. Towards performance analysis of SDFGs mapped to shared-bus architectures using model-checking. In *Proceedings of Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1167–1172, 2013.

[FGFR15] M. Fakih, K. Grüttner, M. Fränzle, and A. Rettberg. State-based real-time analysis of SDF applications on MPSoCs with shared communication resources. *Journal of Systems Architecture*, 61(9):486–509, 2015.

[Fis96] G. S. Fishman. *Monte Carlo : concepts, algorithms, and applications.* Springer, 1996.

[FLE] Flexaware: Real-time streaming analytics data in, action out. `http://www.flexaware.net`. Accessed: 2017-01-12.

[FLM11] M. Fox, D. Long, and D. Magazzeni. Automatic construction of efficient multiple battery usage policies. In *Proceedings of 22nd International Joint Conference on Artificial Intelligence (IJCAI)*, pages 2620–2625, 2011.

[FRA14] Fraunhofer USA. Energy consumption of consumer electronics in U.S. homes in 2013. `https://yearbook.enerdata.net`, 2014. Accessed: 2016-11-28.

[GBS05] M. C. W. Geilen, T. Basten, and S. Stuijk. Minimising buffer requirements of synchronous dataflow graphs with model checking. In *Proceedings of 42nd Annual Design Automation Conference (DAC)*, pages 819–824, 2005.

[Ger14] M. E. T. Gerards. *Algorithmic power management: energy minimisation under real-time constraints.* PhD thesis, University of Twente, 2014.

[GGB+06] A. H. Ghamarian, M. C. W. Geilen, T. Basten, B. D. Theelen, M. R. Mousavi, and S. Stuijk. Liveness and boundedness of synchronous data flow graphs. In *Proceedings of 6th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 68–75, 2006.

[GGS+06] A. H. Ghamarian, M. C. W. Geilen, S. Stuijk, T. Basten, A. J. M. Moonen, M. J. G. Bekooij, B. D. Theelen, and M. R. Mousavi. Throughput analysis of synchronous data flow graphs. In *Proceedings of 6th International Conference on Application of Concurrency to System Design (ACSD)*, pages 25–34, 2006.

[GHH+13] K. Grüttner, P. A. Hartmann, K. Hylla, S. Rosinger, W. Nebel, F. Herrera, E. Villar, C. Brandolese, W. Fornaciari, G. Palermo, C. Ykman-Couvreur, D. Quaglia, F. Ferrero, and R. Valencia. The COMPLEX reference framework for HW/SW co-design and power management supporting platform-based design-space exploration. *Microprocessors and Microsystems - Embedded Hardware Design*, 37:966–980, 2013.

[GHK14] M. E. T. Gerards, J. L. Hurink, and J. Kuper. On the interplay between global DVFS and scheduling tasks with precedence constraints. *IEEE Transactions on Computers*, 64:1742–1754, 2014.

[GK13] M. E. T. Gerards and J. Kuper. Optimal DPM and DVFS for frame-based real-time systems. *ACM Transactions on Architecture and Code Optimization*, 9(4):41:1–41:23, 2013.

[GKA14] C. Gang, H. Kai, and K. Alois. Energy optimization for real-time multiprocessor system-on-chip with optimal DVFS and DPM combination. *ACM Transactions on Embedded Computing Systems*, 13:111:1–111:21, 2014.

[GLMS11] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2010: A toolbox for the construction and analysis of distributed processes. In *Proceedings of 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 372–387, 2011.

[GMA+11] I. Gray, N. Matragkas, N. C. Audsley, L. S. Indrusiak, D. Kolovos, and R. Paige. Model-based hardware generation and programming - the MADES approach. In *Proceedings of 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW)*, pages 88–96, 2011.

[GN00] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software: Practice and Experience*, 30(11):1203–1233, 2000.

[GOO16] Meet Pixel, phone by Google. `https://madeby.google.com/phone/`, 2016. Accessed: 2016-10-17.

[GSB+07] A. H. Ghamarian., S. Stuijk, T. Basten, M. C. W. Geilen, and B. D. Theelen. Latency minimization for synchronous data flow graphs. In *Proceedings of 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD)*, pages 189–196, 2007.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[HG13] P. Herber and S. Glesner. A HW/SW co-verification framework for SystemC. *ACM Transactions on Embedded Computing Systems*, 12(1s):61:1–61:23, 2013.

[HGWB12] J. P. H. M. Hausmans, S. J. Geuns, M. H. Wiggers., and M. J. G. Bekooij. Compositional temporal analysis model for incremental hard real-time system design. In *Proceedings of 10th ACM International Conference on Embedded software (EMSOFT)*, pages 185–194, 2012.

[HKB05] C.-J. Hsu, M.-Y. Ko, and S. S. Bhattacharyya. Software synthesis from the dataflow interchange format. In *Proceedings of 9th International Workshop on Software and Compilers for Embedded Systems*, pages 37–49, 2005.

[HKN15] H. Hermanns, J. Krcál, and G. Nies. Recharging probably keeps batteries alive. In *Proceedings of 5th International Workshop on Design, Modeling and Evaluation of Cyber Physical Systems (CyPhy)*, pages 83–98, 2015.

[HM07] S. Herbert and D. Marculescu. Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. In *Proceedings of ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, pages 38–43, 2007.

[HMGM13] P. Huang, O. Moreira, K. Goossens, and A. Molnos. Throughput-constrained voltage and frequency scaling for real-time heterogeneous multiprocessors. In *Proceedings of 28th Annual ACM Symposium on Applied Computing (SAC)*, pages 1517–1524, 2013.

[HR98] T. A. Henzinger and V. Rusu. Reachability verification for hybrid automata. In *Proceedings of First International Workshop on Hybrid Systems: Computation and Control (HSCC)*, pages 190–204, 1998.

[HRG08] P. H. Hartel, T. C. Ruys, and M. C. W. Geilen. Scheduling optimisations for SPIN to minimise buffer requirements in synchronous data flow. In *Proceedings of 8th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 21:1–21:10, 2008.

[HWZ+12] J. J. Han, X. Wu, D. Zhu, H. Jin, L. T. Yang, and J. L. Gaudiot. Synchronization-aware energy management for VFI-based multicore real-time systems. *IEEE Transactions on Computers*, 61(12):1682–1696, 2012.

[IP05] S. Irani and K. R. Pruhs. Algorithmic problems in power management. *ACM SIGACT News*, 36(2):63–76, 2005.

[JDP08] W. Jang, D. Ding, and D. Z. Pan. A voltage-frequency island aware energy optimization framework for networks-on-chip. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 264–269, 2008.

[JH09] M. R. Jongerden and B. R. H. M. Haverkort. Which battery model to use? *Software, IET*, 3(6):445–457, 2009.

[JHBK09] M. R. Jongerden, B. R. H. M. Haverkort, H. C. Bohnenkamp, and J.-P. Katoen. Maximizing system lifetime by battery scheduling. In *Proceedings of 39th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 63–72, 2009.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.

[Kal04] J. Kallrath. *Modeling Languages in Mathematical Optimization*, chapter Mathematical Optimization and the Role of Modeling Languages, pages 3–24. Springer Publishing Company, 2004.

[Kar78] R. M. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Mathematics*, 23(3):309–311, 1978.

[KCMH10] A. Kumar, H. Corporaal, B. Mesman, and Y. Ha. *Application Modeling and Scheduling*, pages 19–47. Springer Netherlands, 2010.

[KKN96] U. Kiencke, T. Kytölä, and K. J. Neumann. Architectural trends in automotive electronics. *Annual Reviews in Control*, 20:197 – 207, 1996.

[KKZ05] J.-P. Katoen, M. Khattri, and I. S. Zapreevt. A markov reward model checker. In *Proceedings of 2nd International Conference on the Quantitative Evaluation of Systems (QEST)*, pages 243–244, 2005.

[KM66] R. M. Karp and R. E. Miller. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal on Applied Mathematics*, 14(6):1390–1411, 1966.

[KNP11] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Proceedings of 23rd International Conference on Computer Aided Verification (CAV)*, pages 585–591, 2011.

[Koy90] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990.

[KPP08] D. S. Kolovos, R. F. Paige, and F. A. C. Polack. The Epsilon transformation language. In *Proceedings of 1st International Conference on Model Transformation (ICMT)*, pages 46–60, 2008.

[KRPP09] D. S. Kolovos, L. M. Rose, R. F. Paige, and F. A. .C. Polack. Raising the level of abstraction in the development of GMF-based graphical model editors. In *Proceedings of 3rd Workshop on Modeling in Software Engineering (MiSE)*, pages 13–19, 2009.

[KSS+09] J. Keinert, M. Streubühr, T. Schlichter, J. Falk, J. Gladigau, C. Haubelt, J. Teich, and M. Meredith. SystemCoDesigner–an automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 14(1):1:1–1:23, 2009.

[KZH+11] J.-P. Katoen, I. S. Zapreev, E. M. Hahn, H. Hermanns, and D. N. Jansen. The ins and outs of the probabilistic model checker MRMC. *Performance Evaluation*, 68(2):90 – 104, 2011.

[Lam83] L. Lamport. What good is temporal logic? In *Proceedings of IFIP 9th World Computer Congress on Information Processing*, pages 657–668, 1983.

[LEAP95] R. Lauwereins, M. Engels, M. Ade, and J. A. Peperstraete. Grape-II: A system-level prototyping environment for DSP applications. *Computer*, 28(2):35–43, 1995.

[Lee91] E. A. Lee. Consistency in dataflow graphs. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):223–235, 1991.

[Lee93] E. A. Lee. Representing and exploiting data parallelism using multidimensional dataflow diagrams. In *Proceedings of 1993 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, volume 1, pages 453–456 vol.1, 1993.

[LH89] E. A. Lee and S. Ha. Scheduling strategies for multiprocessor real-time DSP. In *IEEE Global Telecommunications Conference and Exhibition (GLOBECOM)*, pages 1279–1283 vol.2, 1989.

[LK10] S. Lee and J. Kim. Using dynamic voltage scaling for energy-efficient flash-based storage devices. In *Proceedings of International SoC Design Conference (ISOCC)*, pages 63–66, 2010.

[LM87a] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 36(1):24–35, 1987.

[LM87b] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. In *Proceedings of the IEEE*, volume 75(9), pages 1235–1245, 1987.

[LMM+16] K. G. Larsen, M. Mikucionis, M. Muñiz, J. Srba, and J. H. Taankvist. Online and compositional learning of controllers with application to floor heating. In *Proceedings of 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 244–259, 2016.

[LPY97] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1:134–152, 1997.

[LSWC08] H. Liu, Z. Shao, M. Wang, and P. Chen. Overhead-aware system-level joint energy and performance optimization for streaming applications on multiprocessor systems-on-chip. In *Proceedings of 20th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 92–101, 2008.

[MAR13] Consumer electronics - global trends, estimates and forecasts, 2011-2018. `http://www.marketresearchreports.biz/analysis/179697`, 2013. Accessed: 2016-08-12.

[MCR01] P. K. Murthy, E. G. Chohen, and S. Rowland. System Canvas: a new design environment for embedded DSP and telecommunication systems. In *Proceedings of 9th International Symposium of Hardware/Software Codesign (CODES)*, pages 54–59, 2001.

[MEM02] Memtime download page. `http://www.update.uu.se/~johanb/memtime/`, 2002. Accessed: 2016-08-09.

[MG93] J. F. Manwell and J. G.McGowan. Lead acid battery storage model for hybrid energy systems. *Solar Energy*, 50(5):399 – 405, 1993.

[Mic96] G. De Micheli. *Hardware/Software Co-Design: Application Domains and Design Technologies*, pages 1–28. Springer, 1996.

[ML02] P. K. Murthy and E. A. Lee. Multidimensional synchronous dataflow. *IEEE Transactions on Signal Processing*, 50(8):2064–2079, 2002.

[MM93] J. F. Manwell and J. G. McGowan. Lead acid battery storage model for hybrid energy systems. *Solar Energy*, 50(5):399 – 405, 1993.

[Moo65] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965.

[Moo97] C. Z. Mooney. *Monte Carlo Simulation*. SAGE, 1997.

[MSH+11] J. L. March, J. Sahuquillo, H. Hassan, S. Petit, and J. Duato. A new energy-aware dynamic task set partitioning algorithm for soft and hard embedded real-time systems. *The Computer Journal*, 54(8):1282–1294, 2011.

[MVB07] O. Moreira, F. Valente, and M. J. G. Bekooij. Scheduling multiple independent hard-real-time jobs on a heterogeneous multiprocessor. In *Proceedings of 7th ACM International Conference on Embedded Software (EMSOFT)*, pages 57–66, 2007.

[NBA16] How to scan your face into NBA 2K17. `http://support.2k.com/hc/en-us/articles/226526407--New-Gen-How-To-Scan-Your-Face-Into-NBA-2K17`, 2016. Accessed: 2016-12-25.

[NM10] N. Navet and S. Merz. *Modeling and Verification of Real-time Systems*. Wiley, 2010.

[NMM+11] A. Nelson, O. Moreira, A. Molnos, S. Stuijk, B. T. Nguyen, and K. Goossens. Power minimisation for real-time dataflow applications. In *Proceedings of 14th Euromicro Conference on Digital System Design (DSD)*, pages 117–124, 2011.

[NPK+05] G. Norman, D. Parker, M. Kwiatkowska, S. Shukla, and R. Gupta. Using probabilistic model checking for dynamic power management. *Formal Aspects of Computing*, 17(2):160–176, 2005.

[OAS15] Oasis face. https://www.keylemon.com/oasis, 2015. Accessed: 2016-12-25.

[Obj12] Object Management Group. OMG Object Constraint Language (OCL), Version 2.3.1, 2012.

[Obj15] Object Management Group. Meta Object Facility (MOF) Core Specification, Version 2.5, 2015.

[OMCM07] U. Y. Ogras, R. Marculescu, P. Choudhary, and D. Marculescu. Voltage-frequency island partitioning for GALS-based networks-on-chip. In *Proceedings of 44th ACM/IEEE Design Automation Conference (DAC)*, pages 110–115, 2007.

[OPT] Optimization vs. simulation [white paper]. https://www.kisters.net/NA/fileadmin/KNA/Products/Optimization_vs_Simulation.pdf. Accessed: 2016-09-19.

[PAD] Software enginering group, University of Paderborn. https://www.hni.uni-paderborn.de/en/software-engineering/. Accessed: 2016-01-14.

[PCL15] S. Pagani, J. J. Chen, and M. Li. Energy efficiency on multi-core architectures with multiple voltage islands. *IEEE Transactions on Parallel and Distributed Systems*, 26(6):1608–1621, 2015.

[Pet62] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Universität Hamburg, 1962.

[PLN92] D. B. Powell, E. A. Lee, and W. C. Newman. Direct synthesis of optimized DSP assembly code from signal flow block diagrams. In *Proceedings of International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, volume 5, pages 553–556, 1992.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of 18th Annual Symposium on Foundations of Computer Science (SFCS)*, pages 46–57, 1977.

[PPL95] T. M. Parks, J. L. Pino, and E. A. Lee. A comparison of synchronous and cycle-static dataflow. In *Proceedings of 29th Asilomar Conference on Signals, Systems and Computers*, pages 204–210, 1995.

[PPS+13] S. Park, J. Park, D. Shin, Y. Wang, Q. Xie, M. Pedram, and N. Chang. Accurate modeling of the delay and energy overhead of dynamic voltage and frequency scaling in modern microprocessors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(5):695–708, 2013.

[Pri92] H. Printz. Compilation of narrowband spectral detection systems for linear MIMD machines. In *Proceedings of the International Conference onApplication Specific Array Processors*, pages 589–603, 1992.

[PSE04] S. Panek, O. Stursberg, and S. Engell. *Optimization of Timed Automata Models Using Mixed-Integer Programming*, pages 73–87. Springer Berlin Heidelberg, 2004.

[QS82] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of 5th International Symposium on Programming*, pages 337–351, 1982.

[Ras05] J.-F. Raskin. *An Introduction to Hybrid Automata*, pages 491–517. Birkhäuser Boston, 2005.

[REC] Recore Systems. `http://www.recoresystems.com`. Accessed: 2017-01-12.

[Rei68] R. Reiter. Scheduling parallel computations. *Journal of the ACM (JACM)*, 15(4):590–599, 1968.

[RPKP08] L. M. Rose, R. F. Paige, D. S. Kolovos, and F. A. C. Polack. *Proceedings of 4th European Conference on Model Driven Architecture – Foundations and Applications (ECMDA-FA)*, chapter The Epsilon Generation Language, pages 1–16. Springer Publishing Company, 2008.

[RPM92] S. Ritz, M. Pankert, and H. Meyr. High level software synthesis for signal processing systems. In *Proceedings of International Conference on Application Specific Array Processors*, pages 679–693, 1992.

[RTP04] A. Rajeev, S. L. Torre, and G. J. Pappas. Optimal paths in weighted timed automata. *Theoretical Computer Science*, 318(3):297 – 322, 2004.

[RVBM96] K. V. Rompaey, D. Verkest, I. Bolsens, and H. D. Man. CoWare-a design environment for heterogeneous hardware/software systems. In *Proceedings of Design Automation Conference (DAC)*, pages 252–257, 1996.

[SAMa] Samsung Exynos 4 Dual 45nm (Exynos 4210). `http://www.samsung.com/global/business/semiconductor/file/product/Exynos_4_Dual_45nm_User_Manaul_Public_REV1.00-0.pdf`.

[SAMb] Samsung Exynos 4 Dual 32nm (Exynos 4212). `http://www.samsung.com/global/business/semiconductor/file/product/Exynos_4_Dual_32nm_User_Manaul_Public_REV100-0.pdf`.

[SAMc] Samsung Galaxy Fame Description. `http://www.samsung.com/uk/consumer/mobile-devices/smartphones/others/GT-S6810PWNBTU`.

[SB09] S. Sriram and S. S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc., 2nd edition, 2009.

[SBGC07] S. Stuijk, T. Basten, M. C. W. Geilen, and H. Corporaal. Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *Proceedings of 44th Annual Design Automation Conference (DAC)*, pages 777–782, 2007.

[SBMP08] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[SC01] A. Sinha and A. Chandrakasan. Dynamic power management in wireless sensor networks. *IEEE Design and Test of Computers*, 18(2):62–74, 2001.

[SDF] SDF3 file format. `http://www.es.ele.tue.nl/sdf3/manuals/xml/sdf/`. Accessed: 2017-01-12.

[SDK13] A. K. Singh, A. Das, and A. Kumar. Energy optimization by exploiting execution slacks in streaming applications on multiprocessor systems. In *Proceedings of 50th Annual Design Automation Conference (DAC)*, pages 115:1–115:7, 2013.

[SGB06] S. Stuijk, M. C. W. Geilen, and T. Basten. SDF^3: SDF For Free. In *Proceedings of 6th International Conference on Application of Concurrency to System Design (ACSD)*, pages 276–278, 2006.

[Sha75] R. E. Shannon. *Systems Simulation - The Art and Science*. Prentice Hall, 1975.

[SKH+97] F. Suzuki, H. Koizumi, M. Hiramine, K. Yamamoto, H. Yasuura, and K. Okino. A HW/SW co-design environment for multi-media equipments development using inverse problem. In *Proceedings of 5th International Workshop on Hardware / Software Codesign (CODES/CASHE)*, pages 153–157, 1997.

[Sny05] J. A. Snyman. *Practical mathematical optimization : an introduction to basic optimization theory and classical and new gradient-based algorithms*. Applied optimization. Springer, 2005.

[SOIH97] W. Sung, M. Oh, C. Im, and S. Ha. Demonstration of codesign workflow in Peace. In *Proceedings of International Conference of VLSI Circuit*, 1997.

[SPRK04] N. Shah, W. Plishker, K. Ravindran, and K. Keutzer. NP-Click: a productive software development approach for network processors. *IEEE Micro*, 24(5):45–54, 2004.

[SRVK10] J. Sprinkle, B. Rumpe, H. Vangheluwe, and G. Karsai. Metamodelling. In *Model-Based Engineering of Embedded Real-Time Systems*, pages 57–76. Springer Publishing Company, 2010.

[SS01] D. P. L. Simons and M. I. A. Stoelinga. Mechanical verification of the IEEE 1394a root contention protocol using Uppaal2k. *International Journal on Software Tools for Technology Transfer (STTT)*, 3(4):469–485, 2001.

[Ste97] R. S. Stevens. The processing graph method tool (PGMT). In *Proceedings of International Conference on Application-Specific Systems, Architectures and Processors*, pages 263–271, 1997.

[Stu07] S. Stuijk. *Predictable Mapping of Streaming Applications on Multiprocessors*. PhD thesis, Technical University of Eindhoven, 2007.

[TCN00] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *Proceedings of 2000 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 101–104, 2000.

[TCWCS92] E. Teruel, P. Chrzastowski-Wachtel, J. M. Colom, and M. Silva. On weighted T-systems. In *Proceedings of 13th International Conference on Application and Theory of Petri Nets*, pages 348–367, 1992.

[Tei12] J. Teich. Hardware/software codesign: The past, the present, and predicting the future. *Proceedings of the IEEE*, 100(Special Centennial Issue):1411–1430, 2012.

[TGB+06] B. D. Theelen, M. C. W. Geilen, T. Basten, J. P. M. Voeten, S. V. Gheorghita, and S. Stuijk. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *Proceedings of 4th ACM/IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, pages 185–194, 2006.

[Tim13] M. Timmer. *Efficient Modelling, Generation and Analysis of Markov Automata*. PhD thesis, University of Twente, 2013.

[TKA02] W. Thies, M. Karczmarek, and S. P. Amarasinghe. Streamit: A language for streaming applications. In *Proceedings of 11th International Conference on Compiler Construction (CC)*, pages 179–196, 2002.

[TKW12] B. D. Theelen, J.-P. Katoen, and H. Wu. Model checking of scenario-aware dataflow with CADP. In *Proceedings of Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 653–658, 2012.

[TUR07] Clarke, Emerson, and Sifakis receive 2007 ACM turing award. `http://www.ams.org/notices/200806/tx080600709p.pdf`, 2007. Accessed: 2016-11-03.

[TY98] S. Tripakis and S. Yovine. Verification of the fast reservation protocol with delayed transmission using the tool KRONOS. In *Proceedings of 4th Real-Time Technology and Applications Symposium (RTAS)*, pages 165–170, 1998.

[Vaa98] F. Vaandrager. *Lectures on Embedded Systems*, chapter Introduction, pages 1–3. Springer Publishing Company, 1998.

[VJ01] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *Proceedings of 2001 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages I–511–I–518 vol.1, 2001.

[VPS90] M. Veiga, J. Parera, and J. Santos. Programming DSP systems on multiprocessor architectures. In *Proceedings of International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 965–968, 1990.

[VSB+13] M. Völter, T. Stahl, J. Bettin, A. Haase, and S. Helsen. *Model-driven software development: technology, engineering, management*. John Wiley & Sons, 2013.

[vvK05] J. van der Tang, H. van Rumpt, and D. Kasperkovitz. HW/SW co-design for SoC on mobile platforms. In *Proceedings of 5th International Workshop on System-on-Chip for Real-Time Applications (IWSOC)*, pages 19–23, 2005.

[WBS07] M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit. Efficient computation of buffer capacities for cyclo-static dataflow graphs. In *Proceedings of 44th ACM/IEEE of Design Automation Conference (DAC)*, pages 658–663, 2007.

[WHL14] E. R. Wognsen, R. R. Hansen, and K. G. Larsen. Battery-aware scheduling of mixed criticality systems. In *Proceedings of 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, pages 208–222, 2014.

[Wig09] M. H. Wiggers. *Aperiodic multiprocessor scheduling for real-time stream processing applications*. PhD thesis, University of Twente, 2009.

[WL85] W. W. Wadge and E. A. Lee. *LUCID, the Dataflow Programming Language*. Academic Press Professional, Inc., 1985.

[WLL+11] Y. Wang, H. Liu, D. Liu, Z. Qin, Z. Shao, and E. H.-M. Sha. Overhead-aware energy optimization for real-time streaming applications on multiprocessor system-on-chip. *ACM Transactions on Design Automation of Electronic Systems*, 16(2):14:1–14:32, 2011.

[WSBL03] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the H.264/AVC video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):560–576, 2003.

[WWDS94] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Proceedings of 1st USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 1994.

[YGB+09] Y. Yang, M. C. W. Geilen, T. Basten, S. Stuijk, and H. Corporaal. Exploring trade-offs between performance and resource requirements for synchronous dataflow graphs. In *Proceedings of 7th IEEE/ACM/IFIP Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, pages 96–105, 2009.

[YTO91] N. E. Young, R. E. Tarjan, and J. B. Orlin. Faster parametric shortest path and minimum-balance algorithms. *Networks*, 21(2):205–221, 1991.

[ZSB+13] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto. Survey of energy-cognizant scheduling techniques. *IEEE Transactions on Parallel and Distributed Systems*, 24(7):1447–1464, 2013.

[ZSJ08] J. Zhu, I. Sander, and A. Jantsch. Energy efficient streaming applications with guaranteed throughput on MPSoCs. In *Proceedings of 8th ACM and IEEE International conference on Embedded software (EMSOFT)*, pages 119–128, 2008.

Titles in the IPA Dissertation Series since 2014

J. van den Bos. *Gathering Evidence: Model-Driven Software Engineering in Automated Digital Forensics*. Faculty of Science, UvA. 2014-01

D. Hadziosmanovic. *The Process Matters: Cyber Security in Industrial Control Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-02

A.J.P. Jeckmans. *Cryptographically-Enhanced Privacy for Recommender Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-03

C.-P. Bezemer. *Performance Optimization of Multi-Tenant Software Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2014-04

T.M. Ngo. *Qualitative and Quantitative Information Flow Analysis for Multi-threaded Programs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-05

A.W. Laarman. *Scalable Multi-Core Model Checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-06

J. Winter. *Coalgebraic Characterizations of Automata-Theoretic Classes*. Faculty of Science, Mathematics and Computer Science, RU. 2014-07

W. Meulemans. *Similarity Measures and Algorithms for Cartographic Schematization*. Faculty of Mathematics and Computer Science, TU/e. 2014-08

A.F.E. Belinfante. *JTorX: Exploring Model-Based Testing*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-09

A.P. van der Meer. *Domain Specific Languages and their Type Systems*. Faculty of Mathematics and Computer Science, TU/e. 2014-10

B.N. Vasilescu. *Social Aspects of Collaboration in Online Software Communities*. Faculty of Mathematics and Computer Science, TU/e. 2014-11

F.D. Aarts. *Tomte: Bridging the Gap between Active Learning and Real-World Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2014-12

N. Noroozi. *Improving Input-Output Conformance Testing Theories*. Faculty of Mathematics and Computer Science, TU/e. 2014-13

M. Helvensteijn. *Abstract Delta Modeling: Software Product Lines and Beyond*. Faculty of Mathematics and Natural Sciences, UL. 2014-14

P. Vullers. *Efficient Implementations of Attribute-based Credentials on Smart Cards*. Faculty of Science, Mathematics and Computer Science, RU. 2014-15

F.W. Takes. *Algorithms for Analyzing and Mining Real-World Graphs*. Faculty of Mathematics and Natural Sciences, UL. 2014-16

M.P. Schraagen. *Aspects of Record Linkage*. Faculty of Mathematics and Natural Sciences, UL. 2014-17

G. Alpár. *Attribute-Based Identity Management: Bridging the Cryptographic Design of ABCs with the Real World*. Faculty of Science, Mathematics and Computer Science, RU. 2015-01

A.J. van der Ploeg. *Efficient Abstractions for Visualization and Interaction.* Faculty of Science, UvA. 2015-02

R.J.M. Theunissen. *Supervisory Control in Health Care Systems.* Faculty of Mechanical Engineering, TU/e. 2015-03

T.V. Bui. *A Software Architecture for Body Area Sensor Networks: Flexibility and Trustworthiness.* Faculty of Mathematics and Computer Science, TU/e. 2015-04

A. Guzzi. *Supporting Developers' Teamwork from within the IDE.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-05

T. Espinha. *Web Service Growing Pains: Understanding Services and Their Clients.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-06

S. Dietzel. *Resilient In-network Aggregation for Vehicular Networks.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-07

E. Costante. *Privacy throughout the Data Cycle.* Faculty of Mathematics and Computer Science, TU/e. 2015-08

S. Cranen. *Getting the point — Obtaining and understanding fixpoints in model checking.* Faculty of Mathematics and Computer Science, TU/e. 2015-09

R. Verdult. *The (in)security of proprietary cryptography.* Faculty of Science, Mathematics and Computer Science, RU. 2015-10

J.E.J. de Ruiter. *Lessons learned in the analysis of the EMV and TLS security protocols.* Faculty of Science, Mathematics and Computer Science, RU. 2015-11

Y. Dajsuren. *On the Design of an Architecture Framework and Quality Evaluation for Automotive Software Systems.* Faculty of Mathematics and Computer Science, TU/e. 2015-12

J. Bransen. *On the Incremental Evaluation of Higher-Order Attribute Grammars.* Faculty of Science, UU. 2015-13

S. Picek. *Applications of Evolutionary Computation to Cryptology.* Faculty of Science, Mathematics and Computer Science, RU. 2015-14

C. Chen. *Automated Fault Localization for Service-Oriented Software Systems.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-15

S. te Brinke. *Developing Energy-Aware Software.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-16

R.W.J. Kersten. *Software Analysis Methods for Resource-Sensitive Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2015-17

J.C. Rot. *Enhanced coinduction.* Faculty of Mathematics and Natural Sciences, UL. 2015-18

M. Stolikj. *Building Blocks for the Internet of Things.* Faculty of Mathematics and Computer Science, TU/e. 2015-19

D. Gebler. *Robust SOS Specifications of Probabilistic Processes.* Faculty of Sciences, Department of Computer Science, VUA. 2015-20

M. Zaharieva-Stojanovski. *Closer to Reliable Software: Verifying functional behaviour of concurrent pro-*

grams. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-21

R.J. Krebbers. *The C standard formalized in Coq*. Faculty of Science, Mathematics and Computer Science, RU. 2015-22

R. van Vliet. *DNA Expressions – A Formal Notation for DNA*. Faculty of Mathematics and Natural Sciences, UL. 2015-23

S.-S.T.Q. Jongmans. *Automata-Theoretic Protocol Programming*. Faculty of Mathematics and Natural Sciences, UL. 2016-01

S.J.C. Joosten. *Verification of Interconnects*. Faculty of Mathematics and Computer Science, TU/e. 2016-02

M.W. Gazda. *Fixpoint Logic, Games, and Relations of Consequence*. Faculty of Mathematics and Computer Science, TU/e. 2016-03

S. Keshishzadeh. *Formal Analysis and Verification of Embedded Systems for Healthcare*. Faculty of Mathematics and Computer Science, TU/e. 2016-04

P.M. Heck. *Quality of Just-in-Time Requirements: Just-Enough and Just-in-Time*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2016-05

Y. Luo. *From Conceptual Models to Safety Assurance – Applying Model-Based Techniques to Support Safety Assurance*. Faculty of Mathematics and Computer Science, TU/e. 2016-06

B. Ege. *Physical Security Analysis of Embedded Devices*. Faculty of Science, Mathematics and Computer Science, RU. 2016-07

A.I. van Goethem. *Algorithms for Curved Schematization*. Faculty of Mathematics and Computer Science, TU/e. 2016-08

T. van Dijk. *Sylvan: Multi-core Decision Diagrams*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2016-09

I. David. *Run-time resource management for component-based systems*. Faculty of Mathematics and Computer Science, TU/e. 2016-10

A.C. van Hulst. *Control Synthesis using Modal Logic and Partial Bisimilarity – A Treatise Supported by Computer Verified Proofs*. Faculty of Mechanical Engineering, TU/e. 2016-11

A. Zawedde. *Modeling the Dynamics of Requirements Process Improvement*. Faculty of Mathematics and Computer Science, TU/e. 2016-12

F.M.J. van den Broek. *Mobile Communication Security*. Faculty of Science, Mathematics and Computer Science, RU. 2016-13

J.N. van Rijn. *Massively Collaborative Machine Learning*. Faculty of Mathematics and Natural Sciences, UL. 2016-14

M.J. Steindorfer. *Efficient Immutable Collections*. Faculty of Science, UvA. 2017-01

W. Ahmad. *Green Computing: Efficient Energy Management of Multiprocessor Streaming Applications via Model Checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-02

Streaming applications such as virtual reality, video conferencing, and face detection, impose high demands on a system's performance and battery life. With the advancement in mobile computing, these applications are increasingly implemented on battery-constrained platforms, such as gaming consoles, satellites, cell phones, automobiles and healthcare systems. Smart scheduling techniques of tasks on processing elements are crucial for self energy-supporting systems where energy harvesting and consumption is kept in balance over the lifetime of the system. These scheduling techniques also help system designers to explore implementation alternatives while maintaining adherence to the performance requirements and battery constraints of a system.