Johan Ersfolk

Scheduling Dynamic Dataflow Graphs with Model Checking

Turku Centre *for* Computer Science

TUCS Dissertations
No 181, August 2014

Scheduling Dynamic Dataflow Graphs with Model Checking

Johan Ersfolk

To be presented, with the permission of the Department of Information Technologies at Åbo Akademi University, for public criticism in Auditorium Gamma on August 15, 2014, at 12 noon.

Åbo Akademi University
Department of Information Technologies
Joukahainengatan 3-5 A

2014

Supervisor

Professor Johan Lilius
Department of Information Technologies
Åbo Akademi University
Joukahaisengatan 3-5, FI-20520 Åbo
Finland

Reviewers

Associate Professor Stavros Tripakis
Department of Information and Computer Science
Aalto University
PO Box 15400, FI-00076 Aalto
Finland

Docent Johan Eker
Ericsson Research
Lund
Sweden

Opponent

Associate Professor Stavros Tripakis
Department of Information and Computer Science
Aalto University
PO Box 15400, FI-00076 Aalto
Finland

ISBN 978-952-12-3091-2
ISSN 1239-1883

Abstract

With the shift towards many-core computer architectures, dataflow programming has been proposed as one potential solution for producing software that scales to a varying number of processor cores. Programming for parallel architectures is considered difficult as the current popular programming languages are inherently sequential and introducing parallelism is typically up to the programmer. Dataflow, however, is inherently parallel, describing an application as a directed graph, where nodes represent calculations and edges represent a data dependency in form of a queue. These queues are the only allowed communication between the nodes, making the dependencies between the nodes explicit and thereby also the parallelism. Once a node have the sufficient inputs available, the node can, independently of any other node, perform calculations, consume inputs, and produce outputs.

Dataflow models have existed for several decades and have become popular for describing signal processing applications as the graph representation is a very natural representation within this field. Digital filters are typically described with boxes and arrows also in textbooks. Dataflow is also becoming more interesting in other domains, and in principle, any application working on an information stream fits the dataflow paradigm. Such applications are, among others, network protocols, cryptography, and multimedia applications. As an example, the MPEG group standardized a dataflow language called RVC-CAL to be use within reconfigurable video coding. Describing a video coder as a dataflow network instead of with conventional programming languages, makes the coder more readable as it describes how the video data flows through the different coding tools.

While dataflow provides an intuitive representation for many applications, it also introduces some new problems that need to be solved in order for dataflow to be more widely used. The explicit parallelism of a dataflow program is descriptive and enables an improved utilization of available processing units, however, the independent nodes also implies that some kind of scheduling is required. The need for efficient scheduling becomes even more evident when the number of nodes is larger than the number of processing units and several nodes are running concurrently on one processor core.

There exist several dataflow models of computation, with different trade-offs between expressiveness and analyzability. These vary from rather restricted but statically schedulable, with minimal scheduling overhead, to dynamic where each firing requires a firing rule to evaluated. The model used in this work, namely RVC-CAL, is a very expressive language, and in the general case it requires dynamic scheduling, however, the strong encapsulation of dataflow nodes enables analysis and the scheduling overhead can be reduced by using quasi-static, or piecewise static, scheduling techniques.

The scheduling problem is concerned with finding the few scheduling decisions that must be run-time, while most decisions are pre-calculated. The result is then an, as small as possible, set of static schedules that are dynamically scheduled. To identify these dynamic decisions and to find the concrete schedules, this thesis shows how quasi-static scheduling can be represented as a model checking problem. This involves identifying the relevant information to generate a minimal but complete model to be used for model checking. The model must describe everything that may affect scheduling of the application while omitting everything else in order to avoid state space explosion. This kind of simplification is necessary to make the state space analysis feasible. For the model checker to find the actual schedules, a set of scheduling strategies are defined which are able to produce quasi-static schedulers for a wide range of applications.

The results of this work show that actor composition with quasi-static scheduling can be used to transform dataflow programs to fit many different computer architecture with different type and number of cores. This in turn, enables dataflow to provide a more platform independent representation as one application can be fitted to a specific processor architecture without changing the actual program representation. Instead, the program representation is in the context of design space exploration optimized by the development tools to fit the target platform. This work focuses on representing the dataflow scheduling problem as a model checking problem and is implemented as part of a compiler infrastructure. The thesis also presents experimental results as evidence of the usefulness of the approach.

Sammandrag

Under det senaste decenniet har vi sett en övergång från allt snabbare enkärniga processorer mot datorarkitekturer med ett ökande antal processorkärnor. Orsaken till den här utvecklingen är att gränsen har nåtts för när varje höjning av hastigheten på en processor orsakar en betydligt större höjning av energiförbrukningen, vilket betyder dyrare beräkningar och samtidigt problem med att processorer värms upp för mycket. Medan utvecklingen inom halvledarteknologi fortfarande erbjuder en ökning av antalet transistorer som ryms på en viss yta, och därigenom erbjuder en motsvarande ökning av beräkningskapasitet som tidigare, är trenden nu att använda dessa till att öka på antalet processorkärnor. Det här har i sin tur lett till att det blivit svårare för programmerare att konstruerar program som effektivt använder den beräkningskapacitet som finns tillgänglig på moderna processorer.

Programmering av parallella arkitekturer anses i allmänhet svårt eftersom nuvarande populära programmeringsspråk i grunden är sekventiella medan parallelismen ofta är något som programmeraren själv inför. Dataflödes programmering har föreslagits som en möjlig lösning för att producera program som skalar till ett varierande antal processorkärnor. Dataflödes beskrivningen är till sin natur parallel och beskriver ett program som en riktad graf där noderna representerar beräkningar och kanter representerar ett databeroende i form av en kö. Dessa köer är den enda tillåtna kommunikation mellan noderna vilket gör databeroenden mellan noderna och därigenom också parallelismen explicit. Genast en nod har sin behövda indata tillgänglig kan noden, oberoende av någon annan nod, utföra sina beräkningar, konsumera indata och producera utdata.

Dataflödesprogrammering har varit intressant inom forskning i flera decennier och har mera allmänt blivit populärt för att beskriva signalbehandlingsapplikationer eftersom grafrepresentationen är en mycket naturlig representation inom detta område. Digitala filter, till exempel, beskrivs vanligtvis med rektanglar och pilar, motsvarande beräkningar och signaler, också i läroböcker. Dataflöde blir också allt mer intressant på andra områden, och i princip är alla program som arbetar på en informationsström lämpliga ett beskrivas med dataflödes paradigmen. Sådana applikationer är bland an-

nat nätverksprotokoll, kryptografi- och multimediaprogram. Som ett exempel har MPEG-gruppen standardiserat ett dataflödesspråk som kallas RVC-CAL för att används inom omkonfigurerbar videokodning för att beskriva videokodnings kompnenter medan kommunikationen mellan dessa komponenter beskrivs som dataköer. Att beskriva en video kodare som ett dataflödes nätverk i stället för med konventionella programmeringsspråk gör programmet mera lättläst eftersom den beskriver hur videodatan flödar genom de olika kodningsverktygen.

Medan dataflöde ger en intuitiv representation för många tillämpningar, uppstår också en del nya problem som måste lösas för att dataflöde ska kunna användas i större utsträckning. Den uttryckliga parallellism i ett dataflödesprogram är beskrivande och möjliggör ett förbättrat utnyttjande av tillgängliga beräkningsenheter, men att ett program består av ett antal oberoende noder innebär också att någon form av schemaläggning behövs. Speciellt när antalet noder är större än antalet beräkningsenheter, vilket betyder att flera noder turvis körs på en processorkärna, blir behovet av en effektiv schemaläggning uppenbart. Det finns flera dataflödes beräkningsmodeller med olika kompromisser mellan uttrycksfullhet och analysbarhet. Dessa varierar från ganska begränsade men statiskt schemaläggningsbara, med minimal schemaläggningskostnad under körningen av programmet, till dynamiska där varje uppgift kräver att ett villkor utvärderas under körningen av programmet. Den beräkningsmodell som används i det här arbetet, nämligen DPN (dataflow process network), som också språket RVC-CAL bygger på, är en mycket uttrycksfull modell som i det allmänna fallet kräver dynamisk schemaläggning under programmets körning. Lyckligtvis möjliggör dock den starka inkapslingen av dataflödesnoder kraftfull analys vilket gör att man kan identifiera delar av ett program som styckvis kan schemaläggas i förväg vilket leder till minskad schemaläggningskostnad under körningen av programmet.

Schemaläggnings problemet består i att hitta de få schemaläggningsbeslut som måste tas medan programmet är aktivt, medan de flesta besluten kan beräknas i design skedet. Resultatet blir då en, så liten som möjlig, uppsättning av statiska scheman där endast valet av schema utförs under programmets körning. För att identifiera dessa dynamiska beslut och för att hitta konkreta scheman, presenterar den här avhandling hur styckvis-statisk schemaläggning kan representeras som ett modellgransknings (model checking) problem. Det här handlar i huvudsak om att identifiera den relevanta informationen som behövs för att generera en liten men fullständig modell som kan användas för att undersöka tillståndsrymden av programmet. Modellen ska beskriva allt som kan påverka schemaläggningen men gömma alla andra detaljer för att undvika att kombinatorisk explosion av tillståndsrymden. Denna typ av förenkling är nödvändig för att göra analysen av tillståndsrymden genomförbar. För att identifiera de konkreta sche-

man i tillståndsrymden behövs en uppsättning schemaläggnings strategier som ger en partiell beskrivning en mängd tillstånd som upprepat besöks medan programmet körs och därigenom kan länkas ihop av några statiska shcheman. På det här sättet kan man göra en uppskattning av de schemans som behövs för att programmet ska fungera korrekt och beskriva dem partiellt så att man kan söka dem i tillståndsrymden.

Resultatet av detta arbete visar att komposition av dataflödesnoder med styckvis-staticka schemaläggningsmetoder kan användas för att anpassa dataflödesprogram för en mängd olika datorarkitekturer med olika egenskaper och ett varierande antal processorkärnor. Detta möjliggör i sin tur att dataflöde kan användas som en mer plattformsoberoende representation eftersom en applikation kan monteras på en specifik processorarkitektur utan att ändra själva programrepresentationen. Istället modifieras programrepresentation av olika utvecklingsverktyg för att passa målplattformen. Arbetet är inriktat på hur man kan representera dataflödesschemaläggningsproblemet som ett modellgranskningsproblem och de verktyg som behövs för att visa att metoden fungerar har konstruerats som en del av en befintlig kompilatorinfrastruktur. Slutligen presenterar avhandlingen några fallstudier med experiment som visar att metoden är användbar och kan användas för att göra program effektivare.

Acknowledgements

One may have a blast working on a dissertation if the work environment is right and the needed support is available. Many-many people and organizations have contributed to this thesis, be it with moral support, technical support, financial support, or pure cooperation.

First of all, I would like to thank my supervisor, **Johan Lilius**, for the opportunity to pursue doctoral studies and to work at the Embedded Systems Lab. for all these years. Speaking of which, I am also grateful for the patience and the appropriate pushing-in-the-right-direction that finally resulted in this thesis. I am also grateful to **Stavros Tripakis** and **Johan Eker** for reviewing this thesis and for providing useful feedback regarding possible improvements to the contents. A special thanks also to **Ream Barclay** who not did not only check the language of the thesis but also helped and instructed me regarding how improve the language of the thesis; after correcting hundreds of issues, the text is already in a readable shape.

The research work related to this thesis including the collection of papers which is part of this thesis, is a result of several peoples' ideas and efforts. I am especially grateful to all the co-authors of these papers, and these are many: **Ghislain Roquire, Marco Mattavelli, Jani Boutellier, Amanullah Ghazi, Olli Silvén, Fareed Jokhio, Wictor Lund, and Johan Lilius**. Also, I would like to thank everyone behind the efforts related to the Orcc project and the large number of CAL applications that are available, as this has been essential for this work to be possible. In particular, I would like to mention **Hervé Yviquel, Mathieu Wipliez, Antoine Lorence, and Mickaël Raulet**, who have always been ready to answer questions related to Orcc and assist with any problems related to the implementation work.

The ESLab has been a great place to work; even if the set of people in the lab has changed during these years and the lab has increased in size, the atmosphere and the working philosophy is still the same as *in the old days*. It is always refreshing to visit the offices of **Stefan Grönroos** and **Dag Ågren** and check-out their latest gadgets. Then with the addition of **Jerker Björkquist** and **Wictor Lund**, the future of technology can not stay unchanged. For the uncountable hours spent discussing the ideas that

now are the contributions of this thesis a huge thanks goes to **Andreas Dahlin** and **Sébastien Lafond**.

There are a large number of issues that are not directly related to research but are needed in order to get the research done. One needs to travel to conferences, get the computers to work, print the dissertation, etc. While the thanks belong to the whole department, I would like to mention the few I personally have felt free to consult at any time, these are especially: **Tove Österroos, Nina Rytkönen, Christel Engblom, Pia Kallio, Joachim Storrank, Niklas Grönblom, Karl Rönnholm, and Marat Vagapov**. I would also like to thank **Tomi Mäntylä** for all the help with getting this thesis printed.

I would also like to thank **IT-department, TUCS, and Tekniikan edistämissäätiö** for funding the my research.

Let us now consider the sequence of events that resulted in this thesis. It all started in 2001 somewhere in the deep forests between Myrkky and Perälä; the names, by the way, mean poison and behind. After roller skating about the kilometers on a gravel road, Richard Westerbäck convinced me that I should join him to Åbo Akademi. So, this we did. Several years later, when it was time to do my masters thesis, again, the same guy, convinced me to join him and do the master's thesis at the Embedded Systems Laboratory. So, an enormous thanks to **Richard Westerbäck** who is responsible for me doing all the right decisions regarding work related questions.

Finally, a big thanks to my family. Being a student for more than two decades requires a lot of patience from everybody around you. First of all I am grateful for all the support from my parents, **Berit and Börje**: *"he ha it fäilast naa"* and to **Mikael, Reetta, Tanja, Abdullah, Yasmine, and Jacob**: *"ni ha heija oåp bra"*. Also to my wife **Tabita** and my family-in-law **Riitta, Mauri, Tomi, and Aili**: *"kiitos kaikesta tuesta"*. And last, not least, but definitely the smallest, the finest of all dogs, **Ruben**, who's contribution to this thesis cannot be described with words.

Åbo, June 17th 2014 

List of Original Publications

1. Johan Ersfolk, Ghislain Roquier, Fareed Jokhio, Johan Lilius, and Marco Mattavelli. Scheduling of dynamic dataflow programs with model checking. In *IEEE International Workshop on Signal Processing Systems (SiPS)*, 2011. Beirut, Lebanon.

2. Johan Ersfolk, Ghislain Roquier, Johan Lilius, and Marco Mattavelli. Scheduling of dynamic dataflow programs based on state space analysis. In *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, pages 1661 –1664, march 2012. Kyoto, Japan.

3. J. Ersfolk, G. Roquier, W. Lund, M. Mattavelli, and J. Lilius. Static and quasi-static compositions of stream processing applications from dynamic dataflow programs. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 2620–2624, 2013. Vancouver, Canada.

4. J. Ersfolk, G. Roquier, J. Lilius, and M. Mattavelli. Modeling control tokens for composition of cal actors. In *Design and Architectures for Signal and Image Processing (DASIP), 2013 Conference on*, pages 71–78, 2013. Cagliari, Italy.

5. J. Boutellier, A. Ghazi, O. Silven, and J. Ersfolk. High-performance programs by source-level merging of rvc-cal dataflow actors. In *Signal Processing Systems (SiPS), 2013 IEEE Workshop on*, pages 360–365, 2013. Taipei, Taiwan.

Contents

Part I

Research Summary

Chapter 1

Introduction

Programming languages and methods are constantly evolving to further improve programmer productivity and make large software projects manageable. This development is driven by the advances in semiconductor design which provide processors and systems with more and more processing capacity, which in turn enable more complex software. Software developers invent new applications that utilize the increasing processing capacity of new hardware, introducing new services that simply were not possible earlier. To enable programmers to develop increasingly complex systems, more advanced tools and methods are developed.

For many years, the main difference between two generations of a processor, for a programmer, was that the uni-processor speed, including the clock rate, doubled every 18 months [12, 96]. Raising the speed of a processor mainly resulted in that the software, without modifications, would run twice as fast on the new processor allowing the developer to add more features to utilize the increased processing capacity. To handle more complex software projects and improve programmer productivity, this development mainly resulted in raising the abstraction level of the programming languages by adding one more level of language constructs on top of the existing ones. In the early history of computers, programming was performed by directly using the instruction-set of the processor; programming using such assembly languages became tedious as the processors gained more processing capacity and the abstraction level was raised to languages where the focus was more on the level of the algorithm than on which instructions to use. Imperative languages, such as C, which have their roots in ALGOL which was introduced in the mid-1950s [15, 101], could be considered to be such languages as the specific instruction set of the processor is not anymore important for the programmer, but the program still reflects a general processor design as every programming construct has a counterpart in the processor instruction set. Such languages are in principle processor independent as the compiler

translates the program into the appropriate processor instructions.

To further raise the abstraction level, introduction of new programming concepts, such as objects, that, instead of being based on processor architectures are abstractions of the concepts programmers typically describe, was studied already in the 1960s, but only became mainstream as late as in the 1990s; partly as a result of the growing popularity of graphical user interfaces for which this kind of abstractions was well suited. This next generation of languages, improved on reusability by collecting functionality and data in a natural way. Object oriented programming has significantly improved productivity and reusability [88], and provides a really intuitive description for some applications, such as, graphical user interfaces, where inheriting the basic functionality, while adding new functionality, hides the less interesting details from the programmer while making it easy to add new features. Further improvements on methods for designing object oriented programs have increased the productivity of the programmer. A natural next step for object oriented methods was to introduce design patterns, that is, more or less, agreed upon general designs that are used to implement certain types software components [92]. All these methods improve the productivity and make large projects manageable, however, they are platform independent only as long as one processor is assumed and do not scale on systems with more processors.

1.1 Recent Developments

Currently, processor clock rates have reached the limit of what is practical with the technology available today and instead of raising the clock rate, the increasing number of transistors on a chip is used to build parallel execution units or processor cores. The reason for this is that while the number of transistors on a chip and the density of transistors continue to increase, the power density on the chip will limit which ones can be turned on at the same time and also the clock rate at which the processor core runs [12]. It can easily be seen that the clock rate of a processor reaches a wall when the frequency and consequently the voltage is increased for a specific technology, by inspecting the switching power of a CMOS transistor $P = \alpha \times C \times f \times v^2$. For different applications or devices, the power is limited for various reasons. For hand-held devises, it is obvious that battery time is a limiting factor, but, also the power dissipation plays a role as it will heat up the device. As an example, in [111, 98], it is said that, the power dissipation for a cell phone, should be kept under 3 W. As a contrast, in data centers, the power consumption directly translates into a cost, when both the power to run the systems and to cool the systems results in enormous electricity bills [90].

This development, while still providing the same increase in processor

capacity, has created an enormous problem for programmer productivity with the current methods for programming and has forced programmers to rethink how to design applications that can utilize many-core systems. Most of the current methods for programming systems with several cores depend on the programmer to make the decision of what to run where, and how the synchronization is handled. The problem with such approaches is that, instead of describing the algorithm, the programmer also needs to think about how to distribute and manage the program on the available system. Furthermore, if the current development continues as predicted, transistors are becoming comparably inexpensive and it will not be possible to use all parts of a chip simultaneously [126, 71]. This means that architectures, already now, but even more in the future, will contain accelerators and specialized hardware of which most parts normally are turned off. As an example, the OMAP 4 platform includes a dual-core general purpose processor, two additional low-power cores, a digital signal processor, a graphics processor, and multimedia accelerators. Using hardware accelerators allows the GPP to run at a lower clock frequency reducing the power consumption [94]. For the programmer, this means that parallelization is not the only problem, but also heterogeneous platforms and the synchronization of the various processing elements need to be handled in an efficient manner.

The potential overheads related to synchronizing complex systems with accelerators can be illustrated by examining the study of the development in energy efficiency of mobile communication equipment between the years 1995 and 2003, presented in [111]. The study in [111] shows that the usage time of such equipment had not improved despite improvements in silicon processes. The development of the hardware had enabled improved functionality and new features to be supported, and with improved energy efficiency. As the complexity of the DSPs and the number of accelerators had grown, the software has become more difficult to manage. In 1995, the software on the phones was still scheduled by the developers in a static manner. This approach was efficient and predictable, the accelerators had deterministic latencies and were used without interrupts and introduced almost no overhead. The more recent models were more complex systems developed by larger teams, and in order to make the development manageable the software was divided into layers and the accelerators were synchronized by interrupts and the scheduling was performed by a preemptive operating system. The ordinary operation of a hardware accelerator is to interrupt the processor when it has finished its task. This interrupt will make the system switch tasks, run interrupt handlers and other parts of the operating system and then switch back to the program using the accelerator. These context switches caused by the interrupts introduce an overhead. Experiments reveal that the impact the context switches have on cache-hit rates significantly affects the overall performance of the system [111]. The per-

formance is further affected by the communication between the processing units introducing an overhead every time an accelerator is used. In order to improve the efficiency, the cache-hit ratios should be increased and the number of context switches kept low. Due to multitasking and sporadic events, static scheduling is not an option. It is impossible to return to systems scheduled by the developers as the systems have grown in complexity. In order to acquire a system with better performance and with less overhead the compilers should have knowledge, not only of the processor, but also of the whole system including the accelerators.

The other direction of hardware development is to make the processor cores more complex and use a number of functional units to utilize instruction-level parallelism (ILP) to increase the parallel operation a program performs when running a sequential program [57]. ILP is a measure of how many operations can be executed in parallel in a computer program. A processor taking advantage of ILP is called superscalar and has multiple functional units executing instructions simultaneously. To be able to exploit ILP, superscalar processors are performing dynamic scheduling of instructions during execution. As the number of simultaneously issued instructions increases, the cost from logic gates required to handle the instructions and check dependencies at run time at the CPUs clock rate increases rapidly. Even if there are no real dependencies between a set of instructions the processor must check for dependencies. In practice there is a limited amount of instruction-level parallelism in programs [119].

One attempt to reduce the overhead of the processors, related to dynamic scheduling, is to move this complexity from the hardware to the compiler; this is called static scheduling [38]. The advantage of static compile-time scheduling is that a compiler has several orders of magnitude more time to make sophisticated scheduling decisions than a dynamic run-time scheduler. Static scheduling performed by the compiler implies that no special hardware is required to analyze dependencies; the result is reduced manufacturing costs and power consumption. The Very Long Instruction Word (VLIW) processors are using this approach, one VLIW instruction encodes multiple operations to be executed in parallel on different functional units of the device. VLIW instructions have as many operation fields as there are functional units on the processor and each field specifies what operation should be executed on the corresponding unit. The compiler examines the dependencies in a sequence of instructions and encodes the operations into instructions containing several operations. To handle the communication inside the processor VLIWs have a forwarding network. The forwarding network is used for bypassing operands from one functional unit to another or for writing operands to a register file. This network becomes complex when the number of functional units grows, and an attempt to further move complexity from the hardware to the compiler is the Transport Triggered

Architecture (TTA) [38, 53], where the program instructions describe how operands are passed between the functional units and register files.

For the programmer, this kind of parallelism is transparent, as it is the compiler or the hardware that finds the parallelism and exploits it. The level of IPL is, however, limited and does not scale to systems with many processor cores. Also, as the number of cores is growing, IPL is really difficult to utilize on several cores as the delays of communicating between cores is big compared to the time it takes to run an instruction. Instead, it is necessary for a programmer to describe larger parts of the program that can be executed in parallel; this again, forces the programmer to construct programs in a different fashion than before.

1.2 Programming Many-Core Systems

Parallel programming has existed for several decades already, and the target of the language constructs has slightly varied with the available platforms, however, the high-level concepts are still rather unchanged. Parallel programs either utilize data parallelism, task parallelism, or a combination of these. Data parallelism can be seen as having many independent data elements for which the computations can be performed concurrently, while task parallelism can be seen as a series of tasks that can be performed in a pipelined fashion. A simple example from the real world should make this clear. If we have a box full of oranges, data parallelism would be to have several people pealing oranges in parallel while task parallelism would be to have one person pealing oranges while another is cutting the oranges into pieces, and a third removes the seeds.

The difficulty with parallel programming is that it does not conform to the real world. If two persons reach for the same orange, it will be clear from the result which one acquired it; with a memory object in a computer program, if two parallel parts of the program grab it, they will both acquire it and when they update it, the result may be a combination of what both of them did, leading to inconsistent or corrupted data. To assure correct behavior, shared data structures must be locked while being modified, however, locking results in waiting and potentially deadlocks. A further problem is that errors may occur sporadically when a program is executed as threads may be interleaved differently each run. Programming with locks is difficult, and requires the programmer to keep track of which lock corresponds to which data; this kind of disciplined work it better suited for a tool than a human being [114].

1.2.1 Current State of the Art

Current parallel programs typically depend on multi-threading as the basis for parallel execution, with the drawback that it is the programmer who needs to express the execution platform (threads) as a part of the implementation. Threads create an illusion of shared memory, however, it is up to the programmer to protect the data against the nondeterministic behavior of using threads as the abstraction for concurrency [84]. As it is up to the programmer to choose when to spawn parallel tasks and when shared data must be locked to prevent race conditions, while it is up to the operating system to choose in which order the threads are interleaved, the programming becomes difficult, error prone, platform dependent, and simply painful to have anything to do with.

One significant drawback with this kind of an approach, is that, the programmer writes the application to fit the parallelism of the target platform. As an example, if the target platform is a quad-core processor, it would seem to make sense to implement a program with four threads co-operating to finish the task. On the other hand, this implementation would not efficiently utilize a new platform with eight cores, as the implementation has been tailored for the previous processor. The other problem is, leaving the synchronization to be solved by the programmer. This is in general very difficult for a human being to get correct. Instead, high-level constructs that describe communication and synchronization, should be available for a programmer, while mutexes and locks should be generated by the tools.

There are extensions to popular programming languages which add primitives for parallel operations. One such implementation of multi-threading is OpenMP, which of a set of compiler directives, library routines, and environment variables, which lets the programmer mark the parts of the code that are meant to run in parallel, while the framework takes care of the thread creation and synchronization. [36]

Methods that have started to gain more interest let the programmer express parallel tasks that can be distributed on the threads/cores by a run-time system. OpenCL (Open Computing Language), for example, is a standard for cross-platform parallel computing. An OpenCL program is typically a program written in a language like C, but with some part of the program, that can run on a separate device, like a graphics processing unit or a digital signal processor, constructed as OpenCL kernels, which are functions written in a flavor of C/C++. The OpenCL kernels are controlled by the host through an application programming interface (API) that are used to define and then control the platforms. [97] OpenCL enables programmers to program heterogeneous platforms without binding the program to much to the platform.

Another problem with using sequential programming languages and adding

parallelism with *glued-on-top solutions* is that the benefit from having more cores is limited by the sequential parts of the program. Amdahl's law [8], describes this relationship as

$$S = \frac{1}{B + \frac{1}{n}(1 - B)}$$

where n is the number of cores, B is the fraction of strictly sequential code, and S is the speedup. This gives an upper bound for how much speedup is possible for a program with a certain amount of parallelism, it also gives the point after which, adding more cores does not improve the speedup anymore.

To avoid unnecessary limitations on the parallelism, a program should be described as parallel as possible; it is not enough that a programmer adds the parallelism needed for one target platform and hopes that it will scale in the future. A number of technologies are available, where the computations are described with explicit dependencies between the calculations. A calculation, then, may, when it completes, enable other calculations that depend on it. The parallelism of the program is then a dynamic property which changes while the program executes, and typically a run-time system is needed to schedule the parallel calculations such that the system is utilized. Such a run-time system typically uses a thread pool to which it distributes parallel tasks, usually by using some kind of dispatch queues.

One implementation of this is the *Grand Central Dispatch* (GCD) which is a technology by Apple Inc. [9], to enable concurrent execution of parallel parts of a program. The parallel tasks which are either functions or a smaller unit which is introduced called blocks, are enqueued and executed on the thread pool of the GCD.

Another example, which also extends the programming model from C/C++ is the Cilk [24] run-time system. With Cilk, a programmer can easily create parallel tasks (called threads in Cilk) which the run-time system distributes on the available processor cores. Cilk is especially good for describing recursive algorithms and a dependency graph is constructed at run-time as a consequence of (recursive) function calls in the program code. This graph, which is a directed acyclic graph, describes the dependencies between the tasks of the application. A newer version of Cilk called Cilk Plus is currently part of Intel Parallel Studio, which is a toolkit by Intel Inc. for parallel programming.

For other types of applications such as multimedia and signal processing algorithms the dependency graph is more static and can therefore be described statically as a dataflow network, where the dataflow nodes describe tasks and the edges data dependencies.

1.2.2 Dataflow programming

Dataflow has been proposed as one solution for how to represent parallelism of an application. A dataflow program consists of independent computational nodes only connected by communication queues. As such the nodes can run in parallel as long as there is enough input and enough space on the output queue. Dataflow programs explicitly express the parallelism of the program and the nodes can easily be distributed to many processor cores. This is guaranteed by the strong encapsulation of dataflow actors as the actors are not allowed to share state but only communicate through directed connections. [21]

Dataflow programs provide a parallelism which resembles task parallelism, as actor firings can be seen as tasks that can execute in parallel. The programmer describes firings, the operations that should happen, the input and output rates, and possibly a condition enabling a firing to take place. A dataflow program scales to platforms with different degrees of parallelism, with some restrictions, depending on how fine grained the dataflow implementation is. While dataflow programs provide a flexible parallel implementation, this does not automatically mean that the program will be efficient. As the control flow is not explicitly described, but instead the concurrency is specified, scheduling is required when several actors shares a processor.

Dataflow programming has proved itself useful in signal processing applications as it maps nicely to the mathematical representations used within this field. It is common praxis to represent filters and transforms as boxes connected with data queues and many tools used for modeling use a dataflow representation, e.g. Simulink and Labview. However, for dataflow programming to be useful for a wider range of applications, the programming methods and tool support need to improve.

For the type of applications, for which dataflow typically is used, performance of the generated program is important for how useful the application will be. For dataflow programs, a central aspect that has a significant impact on the performance of the program is the scheduling of the dataflow nodes. On a theoretical platform with more cores that dataflow nodes and no communication overheads, a dataflow program does not need scheduling as every node simply occupies one core and waits for input to arrive. On a real platform there are usually more dataflow nodes than processor cores and, in the case there actually are more processor cores, the communication overheads between the cores make it necessary to map several nodes to the same processor core. This means that scheduling, that is, deciding what to run next, is required.

1.3 The Problem Statement

Dataflow programming can be used to highlight the parallelism of an application. A dataflow program is a directed graph, where arcs represent data dependencies between the computational nodes; as any other communication is forbidden, the communication and dependencies between the nodes are explicitly described. For a dataflow program to be truly platform independent, it should adapt to both platforms with massive parallelism as well as to single processor systems, but it should also be possible to tune it for systems with different cache sizes and for processors with varying penalties for branch instructions. This means that the size of the computational nodes (actors) must adapt to the target platform, either by merging smaller nodes into larger or by splitting larger nodes into a set of smaller; these operations affect the scheduling of the nodes but also the size of data a node works on during one firing.

The approach chosen for this work requests the programmer to construct an as fine-grained as possible design, which then is transformed by actor composition in to something that is efficient on the specific target platform. The composition implies a joint scheduling of the actors, which can be made efficient by removing some generality from the composition by removing some of the scheduling decisions. Such a simplified composed actor, again, needs to be proven correct in that it still accepts every possible input stream that the original actors did. Also, deadlock freeness must be proven for a program after composition of actors.

This thesis discusses approaches for choosing appropriate program partitions for composition, such that it is possible to reason about the efficiency and correctness of the scheduler of the composition. The thesis includes discussion about the different aspects of a scheduling model that relates to the correctness of the model and shows how the complexity of a scheduler can be approximated. A full prototype tool chain, that takes a CAL program, performs partitioning, scheduler composition, and actor merging, is presented.

As some of the definitions used in this work may have different meanings in other contexts, it is necessary to define what is intended with some of the central concepts discussed in this thesis.

Composition Actor composition is used to transform a set of actors into one single larger actor with less scheduling overhead. In this context, composition means that the action schedulers of a set of actors are composed in to a single scheduler. The goal with the composition is to allow a simplified action scheduler and the composed scheduler is not required to allow all the different firing sequences that was allowed by the original program, however, it is required to produce identical outputs for an input sequence.

Scheduling The composed scheduler is simplified by having as many of the scheduling decisions as possible done at compile-time. The scheduling is about finding the sequences of action firings that always follow each other. The composed scheduler runs such sequences, called static schedules, instead of single actions. In practice this means that the scheduling sequentializes a set of actors. Scheduling may also refer to the run-time scheduling which includes choosing and running one of the composed schedulers, however, it should be clear from the context which type of scheduling is discussed.

Merging The actual process of creating a new actor from the composed actor scheduler and the functionality of the actors that has been jointly scheduled is called actor merging. The actor merging decides how the functionality and data channels of the different actors are put in to a single actor, and how the composed scheduler is integrated in the actor. The result of the merging is then a new actor that can be code generated instead of the original actors.

1.4 Contribution of the Thesis

The problem of scheduling dynamic dataflow programs is manifold and can be explored from several angles. Starting from a choice between different models of computations, which restricts or enriches the model with extra information, to approaches which try to fit parts of a program in to more static descriptions, and by this, extract static schedules. The approach investigated in this thesis, starts from an expressive model, which is allowed to describe dataflow actors with any behavior. These actors are analyzed and a model describing the behavior of an actor partition, with a limited state space only including scheduling related information, is constructed. Then the state space is analyzed, and static schedules that link some of the reoccurring states of the program are extracted and used to generate a quasi-static scheduler. The problems discussed in this thesis are different aspects of how this analysis is made feasible.

The first thing that is needed to even make a state space analysis of a dataflow program feasible is to reduce the state space to only include the information essential for the scheduling analysis. A CAL program, essentially, is a set of dataflow actors which can be described as state machines with variables, and which are connected by data channels. One of the main points in the thesis is therefore **to show how to describe the state space of a program, with the minimal information such that the program can be scheduled correctly.** This property is a central requirement in the work presented and is therefore discussed throughout the thesis, but mainly in Papers 2 and 4, and in Chapters 3, 4, and 5.

A related issue, which is important for constructing a set of guards for a composition of actors, relates to the set of firing rules that are used by the quasi-static scheduler, and how these can be chosen. The related research problem is how one can **show that a set of guards are strong enough to describe the behavior of a partition, and how control values should be modeled in order to choose suitable partitions for composition.** This is on a more abstract level discussed in Chapter 3 and with some more details in Paper 4; then in Chapter 5, more concrete examples are given of how this issue can be handled.

With a model that efficiently describes the state space of the program and correctly gives an adequate set of guards that describes the operations of a dataflow program, or part of one, the next problem is to actually extract both the static schedules and a scheduler which describes how the schedules should be fired based on the set of guards provided. The thesis, therefore, also **presents a number of strategies that can be used to find static schedules out of the state space of a program partition by using a model checker.** The initial work related to this issue is presented in Paper 1, however, a more in depth discussion is provided in Chapter 5, while some case studies are presented in Chapter 7.

With the composition and partial compile-time scheduling of the actor partitions successfully completed, one would hope that the transformed program in every sense performs better than the original program where each actor is individually scheduled at run-time. Now, this is unfortunately not simply a software issue, as the processor architecture plays a crucial role in deciding how a composition and a specific schedule impacts on the performance. For this reason, the thesis **contributes with measurements and conclusions regarding how partitioning, scheduling, and composition affect the performance of a program.** Measurements are mainly presented in Papers 3 and 5, while Chapter 6 provides a more general discussion of this topic.

The work done related to this thesis shows that this type of methods can be used to schedule CAL actors for composition. However, to enable more automatized and successful scheduling of CAL program, some improvements regarding specification of programs could be useful for a future tool set to allow easier verification of properties related to scheduling, but potentially also others. The case studies in Chapter 7 highlights some constructs that make the scheduling difficult, some possible solutions on how to implement dataflow programs or how specifications could be added to a program is discussed in Chapter 8.

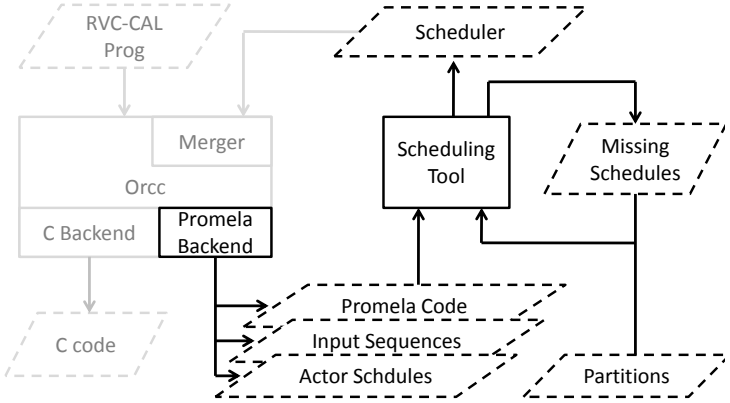

Figure 1.1: The artifacts produced for the scheduling task and how these connect to the existing tool chain (gray).

1.4.1 Constructed Artifacts

In order to evaluate the proposed approach and to measure the impact this kind of composition and scheduling has on a dataflow program when it is mapped on to different platforms, it is necessary to build a number of software artifacts that can be used for this purpose. The scheduling approach is implemented as part of a existing compiler infrastructure and some additional external tools, which produce interchange artifacts that describe different aspects of the dataflow program. Figure 1.1 illustrates the main software components and the information that is passed between these. A central part of the tool chain is the Orcc compiler which is both used to generate the representation of the program that is used in the scheduling as well as to finally generate the executable program, based on the information generated by the scheduling tool, to a language such as C, that can be compiled to the target platform.

The approach is described in more detail in the following chapters, however, to put these in a context, some of the high level ideas and key artifacts should be introduced.

Scheduling Tool The scheduling tool, which consists of a model checker and a wrapping software that specifies the model checking tasks and analyses the model checking result. This software uses information generated from the dataflow program description and defines a set of schedules for the model checker to search for. The scheduling is a iterative task where one found schedule may introduce new schedules to search for, this is illustrated by the feedback loop of the scheduling tool in Figure 1.1. This part of the

scheduling approach is mainly described in Chapter 5.

Partition The set of actors that are to be composed and thereby analyzed for scheduling is called a partition. A partition may correspond to the whole dataflow program but usually partitions are chosen, by the designer, to minimize the scheduling overhead by grouping together actors with related behavior, and to improve run-time performance by choosing partition sizes that fit the target platform. Partitioning is only described in this thesis from the point of view of correctness, where correctness means that a model describing the scheduling of a partition must include all the necessary information, while the performance aspect is left for an external design space exploration. The required properties of a partition is discussed in Chapter 3, which presents the different types of dynamic behavior a partition of actors can have.

Schedules The task of the scheduling framework is to produces static schedules, that is, sequences of firings that, once chosen for execution, the whole schedule can be guaranteed to run to completion without the need for evaluating guard expressions. When several schedules are need to describe the behavior of a partition, the choice of schedule is performed by a scheduler at run-time. The scheduler produces, the schedules and the firing rules for these, is then the artifact that is the end result of the scheduling tool and is used by the compiler to merge actors before the code is generated for the target platform. The construction of a set of static schedules for a partition together with a scheduler that corresponds to the firing rules of the schedules, is mainly discussed in Chapter 5.

To construct and initialize the model checker for one scheduling task a few pieces of information, either automatically generated from the dataflow program or provided by the developer, is need.

Promela Code The behavior of the dataflow program is translated to the language used by the model checker, in this case Promela. For this purpose a backend producing Promela code is constructed for the Orcc compiler. The generation of the Promela code not as such interesting in this context, and is only briefly described in Chapter 5 and Paper 1, instead finding the relevant information to generate code from is more relevant. An addition to generating the Promela code, the backed runs a set of analysis passes which are used to simplify the program description to only include scheduling information as well as producing additional information which is used by the scheduling tool, this involves information regarding static schedules and their corresponding input sequences of individual actors.

Actor Schedules Simplifying of the individual actors to only include input dependent firing rules, while firing sequences that can be resolved from the program text of the actor are described as static schedules, gives a good staring point for the actual scheduling task. Generating a set of static schedules that describe the possible firing sequences of the actor, gives a predictions of what the static schedules, of a partition, that are to be searched for by the model checker should look like. Similarly, it also gives the data rates and the needed input sequences that are consumed by the corresponding schedules.

Input Sequences A final artifact that is produced as an input for the model checking task, is the set of input sequences an actor accepts. These input sequences correspond to the per actor static schedules described above, and describe the number of data tokens a schedule consumes and, if it can be resolved, the value of a control token that enables the schedule. This information is used to initialize the model checking tasks as well as to decide when a schedule has finished based on the number of tokens it has consumed. The construction of the actor schedules and the corresponding input sequences is discussed in Chapter 4.

1.5 Research Methods in this Thesis

This thesis presents research related to the scheduling problem of dynamic dataflow programs, where the dataflow programs in practice are programs implemented using the Cal Actor Language (CAL) [46]. The research starts from an idea that the dataflow program can be modeled as a set of communicating finite state machines (FSM), for which a state space analysis can give the set of schedules that describes the behavior of the program. Apart from building tools and prototypes for evaluating the ideas, the research also includes performing measurements evaluating the impacts of the approach and algebraic analysis of the correctness of the models.

To narrow down the discussion on research methods, the discussion is based on the book *On Research Methods* [74] by P. Järvinen, and the individual parts of the research work is compared to the methods presented in this book.

1.5.1 Design-science research

A great part of the work presented in this thesis relates to tools and artifacts that have been built to solve the problems of scheduling dataflow applications. This kind of a research problem fits well with design-science, which describes a methodology for research where the target is to create new innovative artifacts for a specific problem domain, and, while this method

has not strictly been followed, it is relevant to compare the presented work with the main points of this research method.

According to March and Smith (1995) [91], design-science has two basic activities, *build* and *evaluate*, and four types of products or artifacts, *constructs*, *models*, *methods*, and *instantiations*.

The **build** activity corresponds to building tools and methods for solving a specific problem, and the build activity is performed to construct a prototype that shows that the proposed idea is feasible and enables a evaluation of the proposed approach. What separate design-science research from engineering is that the construction does not follow what is best praxis in the field, but instead, the purpose of building an artifact is to create new knowledge and to enable an evaluation of a new idea. It is therefore important in design-science that research resources are not used to build artifacts which are based on already known and evaluated ideas. [91, 74]

A constructed artifact then needs to be **evaluated** in order to answer the question regarding if the artifact contributes to the progress of the field. The questions that needs to be asked are: How well does the artifact perform the task? How does it compare to previous work? To be able to evaluate the artifact, some criteria for success and metrics need to be defined, such that results are comparable. Of course, if no previous methods have been able to achieve the same goals, the feasibility of constructing the artifact is already a valid result. [91, 74]

The building process related to this thesis, includes extending a code generation tool chain with a set of analysis and transformation operations. To evaluate how well the artifacts work, some metrics that can be compared are: how fast the generated code is when run on different platforms, how usable the tool is regarding how large parts of a program can be transformed in to something more efficient, and finally, how usable the tool is for a developer, where usable means how difficult the tool is to use and how much time it requires. These metrics can then be compared to previous work, however, some criteria for success are also needed to define what can be regarded as success. In this case, it is not required that the usability improves, but instead, it is a trade-off with the other metrics.

Design-science research must produce an artifact, according to the definitions of March and Smith, we have the four types of artifacts as follows.

The first one, **Constructs**, provides the language in which problems and solutions are defined and forms a vocabulary for the domain [91, 74]. According to March the evaluation of constructs typically involve completeness, simplicity, elegance, understandability, and ease of use. In the dataflow domain, which is rather mature, the *constructs* have been developed over several decades and for this reason adding new constructs can typically be avoided. In this work, existing constructs are used to keep the discussion simple and not introduce overlapping constructs; instead, the focus is on

other types of artifacts.

Models are built as a set of proposition or statements describing relationships between the *constructs* and are used to improve the understanding of the problem and solution and ties the approach to the real world problems. According to March the models are evaluated in terms of their fidelity with real world phenomena, completeness, level of detail, robustness, and internal consistency. In this work, several models are constructed which each highlights one property of the dataflow program that is being analyzed. These methods are for this reason presented formally to demonstrate the properties related to completeness and consistency, as an incomplete model in the context of code analysis and transformation, is worthless. The evaluation of the models is presented in Paper 4, and Chapter 3.

The **methods** are algorithms or guidelines that describe how a specific problem should be solved, or more applicable in the context of this thesis, how to search the solution space. According to March the evaluation of methods mainly regards their efficiency, generality, and ease of use. In the work of this thesis, this relates mainly to strategies related to scheduling, how the information from the models is used to perform the actual scheduling. The evaluation of methods is mainly presented as case studies in Chapter 7 of this thesis.

Instantiations are the actual tools constructed to demonstrate the approach and show that constructs, models, or methods can be implemented in a working system. The instantiations are the artifacts that link researchers to the real world and show how the artifacts react to it and how users react to the artifact. According to March the evaluation of instantiations relates to the efficiency of the artifact and what impact it has on the environment and users. For the tools produced within this work, this would mean that a number of developers should be able to use the tools and feel that it helps them achieve some of their goals.

Evaluation of the Work The research work behind this thesis was not directly based on design science, but because of many similarities regarding the goals of the research and the kind of experiments needed, it is relevant to evaluate this work as a design science research problem. This can be done by analyzing the seven guidelines, regarding design science research, given by Hevner et al. in [65].

The first one, *Guideline 1: Design as an Artifact*, is obviously followed as described above according to the definitions of March and Smith. The second guideline, *Guideline 2: Problem Relevance,* states that these artifacts should provide solutions to important and relevant business problems. In this context, this means that the problem that is solved has a value in industrial applications and that it aids to the development in the field. The relevance cannot be evaluated from the research work but instead from the

problem statement motivating the research. The relevance of the scheduling problem comes from the fact that, using dataflow in many industrial applications is avoided due to the lack of methods for generating efficient code.

Guideline 3: Design Evaluation. It is crucial that the resulting artifacts are evaluated with proper metrics and sufficient experimental data. Within this context, when the artifact is part of a design tool chain, the relevant properties are related both to the usability of the artifact from the point of view of the user of the artifact, but also regarding how the artifact affects the performance of the dataflow program which is transformed by the artifact. For the user, the question is how much manual labor and additional expertise is needed to use the artifact and how much time the user must wait for the tool to finish. These properties are evaluated in Chapter 7 which provides a set of case studies where the artifact is used to construct schedulers for composed actors. While several of the studied examples are automatically transformed by the artifact, the ones that are not are also important for giving an improved understanding of the research problem. For the second part, regarding the performance of the actors that are composed by the artifact, the evaluation requires numerous experiments including different configurations and target platforms. Experimental data is presented in Chapter 6 and in several of the original publication, however, what is even more important than being able to show promising numbers is that the experiments are properly performed. We will get back to evaluating the measurements shortly, after discussing the other guidelines by Hevner et al.

Guideline 4: Research Contributions. Following the discussion on design evaluation, a more general evaluation is defining clear and verifiable contributions of the work. According to Hevner et al. the contribution can be the design artifact itself or contributions related to foundations and methodologies if the area of the design artifact. While much of the work presented in this thesis evaluate experimental strategies for building scheduling models and deriving schedules for composed actors, the design artifact both shows that such methods are implementable but also enables an evaluation of methods presented. The following guideline, *Guideline 5: Research Rigor,* relates to the research contributions by requesting rigorous methods in both the construction and evaluation of the design artifact [65]. In the context of this work, rigor relates to the set of dataflow programs that are used to design and evaluate the artifact such that the artifact can be shown to solve a general problem and not only a special case. Similarly, rigor can be related to the data sets that are used to perform experiments and evaluation of the artifact; rigorous experiments make the knowledge acquired from the experiments more general. Furthermore, to show that the presented methods are complete, with respect to the possible inputs to the artifact, the methods need to be described mathematically based on an formal description of the

dataflow program.

The next guideline, *Guideline 6: Design as a Search Process,* defines the research as an iterative process, where available *means*, which are resources available to construct a solution, are used to reach desired ends, representing goals or constraints on the solution, while satisfying laws in the problem environment [65]. For complex problems, the first version of an artifact typically requires simplifications of the problem space or decomposition in to subproblems. While such an artifact hardly can be expected to as such be usable, it enables an improved understanding of the problem and raises relevant questions. The last guideline, *Guideline 7: Communication of Research,* elaborates this idea by highlighting the communication of the research results to the relevant audience. The important result of the work is then not the artifact itself but the knowledge acquired from constructing the artifact and by conducting the experiments enabled by the artifact.

1.5.2 Experiments and Measurements

One aspect of evaluating the approach is to construct experiments and perform measurements. In the context of CAL, relevant experiments are multimedia applications such as video decoders and various signal processing applications like network protocols and digital filters.

For measurements to make sense, it is essential to plan what is actually measured and what parameters may affect the measurement. To draw more general conclusions from the measurements, it is also important that the experiments are repeated for several application, input sequences, compiler settings, and hardware configurations. In Paper 3, where a larger set of experiments is presented, it is obvious from the results that, had the experiments been done for only a part of the chosen experiments, the results would not have given a realistic view of the impact of the presented program transformations. The reason is that the same configurations give, not only different, but contradicting results on different platforms that were used in the experiment. Instead of simply concluding that a transformation of a program is good or not, it is then possible to find how platform parameters such as cache size, has an impact on how the transformation affects the program performance. To be able to make a conclusion, it is therefore important to have a large enough set of test cases.

Measurement Errors and Variation Performing measurements on a computer system is made complicated by operating systems, caches, and advanced processor features. In general, it is hard to know if a specific measurement is a result of the feature being investigated or if it is a coincidence of a combination of features that are not taken into account in the measurement setup. Further, the same experiment may give varying results for every

run; and in some cases, already by touching the mouse of the computer, the samples measured during that interval may be quite far from the average. To make the results valid, basic statistical methods, such as calculating a confidence interval, can be used. This way, it is possible to say if a result is statistically significant or not.

What we are interested in when measuring the performance of the dataflow programs after composition, is the speedup compared to the original program or other configurations. The speedup can be defined as the difference between the time it takes to run a program to completion, or in case the program does not run to completion, the difference between the rate at which the programs complete a tasks e.g. the frame rate of a video application.

A useful measure of the speedup of an application is the mean value accompanied by an confidence interval. The confidence interval gives an interval within which a sample will reside with a given probability (e.g. 95%). When measuring two configurations of a system, if the confidence intervals of the two configurations do not overlap, the measured difference can be considered to be significant. In order to calculate a confidence interval, however, the measurements, and the variation of these, must be assumed to follow a normal distribution. Measuring a computer system, there are many possible sources of both systematic and unexpected errors. In the measurement related to this work, measurements were performed according to the guidelines by Lilja in [86].

Especially some errors in the measurements are more difficult to handle, as an example, if the operating system decides to run a task during one of the measurements; this measurement may then be completely useless and misleading. For this reason, the samples that clearly are outliers should be removed from the measurements to assure that the samples have a normal distribution. The removal of outliers, however, must be justified. Outliers on the sample set of the measurements of the speed of a program typically are single values with much worse results than the other samples which reside in a comparably small interval.

When the measurements are properly handled and the different configurations can be compared, the next important thing is to be able to draw conclusions from the result, or more importantly, to draw correct conclusions.

Generalizability of Results A property we are interested in is the generalizability, which refers to how useful a theoretical construct is outside the limited set of observations from which it has been constructed [74]. It would be tempting to state, based on the experiments, that actor composition results in faster code, however, does the set of example programs, the set of platforms, the different configurations, and the measurements themselves, justify this generalization? Fortunately, the experiments already show that

this is not the case; instead it can be concluded that composition is a useful part of design space exploration.

1.6 Structure of Thesis

This thesis is constructed as a collection of peer-reviewed conference papers which are accompanied by a more general introduction to the field and an overview of the topics presented in the papers. The two parts of the thesis are to some extent overlapping, with the difference that the first part attempts to explain the concepts and background in a more general fashion while, in the second part, the papers are more technical and concentrate on narrower subjects within the research. As a result, the two parts are also independent and part one gives an understanding of the research work without the need to study the more detailed papers.

The first two chapters of the thesis is an introduction to parallel programming and dataflow, and the purpose of these is to motivate why the research is needed and describe the relevant concepts that the later chapters depend on. In the second chapter, work regarding dataflow and process models is presented, introducing the problems related to scheduling and properties such as boundedness. Chapter 2 also presents the CAL Actor Language, which is used as the programming language to implement the dynamic dataflow programs that are studied.

Chapter 3 is a discussion of the dynamism that can be implemented by CAL actors and the point with the chapter is to show what a real dynamic scheduling decision is and which scheduling decisions can be resolved by analyzing the context of the actor. The chapter discusses properties such as input dependency, non-determinism, and monotonicity. Methods for reasoning about such properties is then presented; this is partly based on Paper 4 [49] with some additions of previously unpublished work.

The following two chapters present actual scheduling methods. First, Chapter 4 presents the scheduling of a single actor by removing scheduling decisions that can be resolved from the information in the actor program text. Then Chapter 5 presents how the scheduling can be further resolved by analyzing a partition of actors such that some dynamic behavior becomes static when the context of the actor is known. Chapter 5 is partly based on Paper 1 [51] and Paper 2 [52], when it comes to generating the actual schedules, and Paper 4 [49] regarding the analysis of the actor partitions. Furthermore, Chapter 4, contains work that has been implemented and is needed by the methods presented in the following chapter, but has not previously been published.

Finally, Chapters 6 and 7, evaluate the result and draw some conclusion regarding the presented approach. Chapter 6 presents measurement results,

partly published in Papers 3 [50] and 5 [26], and then discusses these results and what further steps, regarding design of languages and tools would be needed to achieve better results and possibilities for analysis and verification. Chapter 9 then presents the papers and some final conclusions are presented in Chapter 10.

Chapter 2

Some Background Regarding Dataflow Models

Ideally, a computer program describes an algorithm or a set of computations without making any assumptions about the underlying hardware and without imposing unnecessary restrictions on how, or in which order, the computations are to be performed. It is instead the compiler's job to make the appropriate choices that are needed for the program to run efficiently on the desired hardware, but without altering the results computed by the program. With parallel programming, this becomes even more relevant as unnecessary restrictions, imposed by the programmer over specifying the program, limits the available parallelism.

For this reason, many computational models for describing concurrent computational entities, have been proposed and have to some extent been used in industrial strength tools. In digital signal processing, programs are often described as directed graphs where edges describe streams of data samples and nodes describe calculations performed on these samples. This kind of applications are typically represented with a *visual syntax* to specify the graphs, e.g. Ptolemy from the University of California at Berkeley [29] and MATLAB from MathWorks with its visual interface SIMULINK, which makes the description intuitive for humans to read or use readymade components to build models. While a graphical representation is of no consequence for a compiler, the graph representation explicitly describes the parallelism of the program.

Programming models based on the dataflow paradigm have a long history starting with work form the early 1960's. One of the *early* languages, was presented by Dennis in 1974 [42]. This dataflow language describes program functionality as a bipartite directed graph with two types of node, namely *links* and *actors*. A number of actors are provided, which either operate on control or data values (tokens) or perform a conditional operation on

some data values depending on a control value, and links which are one token queues and distribute the token to one or more other nodes. For a software engineer, the language presented resembles logic gates schematics, however, the nodes implement firing rules which means that the semantics compared to a Boolean function is quite different. In Dennis's language links are also called nodes as these also have firing rules, e.g. links that distribute one input token to several nodes. In this thesis, however, when a node is mentioned it corresponds to an actor, links on the other hand are referred to as the edges of a dataflow graph.

In addition to Dennis's dataflow language, early work was done by Petri who presented Petri nets 1962 [102], Estrin and Turn who presented a dataflow model in 1963 [54], Karp and Miller who presented computation graphs in 1966 [77, 78], Chamberlain who presented a single assignment dataflow language in 1971 [35], Kahn who presented a process language in 1974 [76], and in 1977 Arvind and Gostelow [10, 11], and Gurd and Watson [63] presented their separate works on tagged token dataflow models. But also more recently, dataflow languages have started to attract more interest and languages such as StreamIT [117] and CAL [46], has been presented. Also, much work has been put into developing different computational models with a trade-off between expressiveness and predictability, and many of these will be presented later in this chapter.

Many programming languages and models for describing this type of applications exist, and have been developed for the last half century; a thorough survey is given by Johnston et al. in [75]. Many contemporary dataflow tools and languages have their roots in, or at least much in common with, Process Calculi [56, 118, 95, 64], the Actor Model [66, 13, 37], and/or Data Flow Models [42, 82]. While these models have much in common, and can be seen as concurrent processes exchanging messages, still, the different models have some fundamental differences and provide different primitives for concurrent processing. The languages typically consist of a *host language* describing the behavior of the nodes and a *coordination language* describing the interaction between nodes [83]. What primitives are allowed in each of the models and which model a languages depends on inherently decide what can be expressed by the language and which properties the resulting program has regarding predictability and determinism. Before going into more specific scheduling approaches, it will be appropriate to present some of the basic models and the properties of these.

2.1 Processes, Actors, and Dataflow Models

Communication between processes has been studied for quite some time, and in a more general context than only streaming applications. Some of

the significant initial work on *Cooperating sequential processes* was done by Dijkstra in 1968 [44], where synchronization primitives such as semaphores are presented. For dataflow (or related) models, the abstraction level is raised to deal with communication on the level of massages and queues, and primitives such as semaphores, monitors, threads, etc. which may be used to implement the program in practice, are hidden for the programmer within higher level constructs. This is perhaps the most significant commonality the models of interest share on a practical level, but of course, the exact primitives available and the semantics of these is also what makes the difference between these models.

In the following, some more general models, which are of consequence for this thesis, are presented. Then in the following sections, more specialized dataflow models of computation and the properties these provide are presented, and finally the CAL actor language is presented.

Communicating Sequential Processes Hoare presented in 1978, *Communicating Sequential Processes* (CSP) [69], which adds a set of primitives for managing creation of and communication between concurrent processes and combines this with Dijkstra's Guarded Commands [45] to describe the behavior of the processes. CSP has been useful especially for verification of concurrent properties of different systems [17], and has significantly evolved during the years since it was presented. For a more contemporary version see [70], however, here we concentrate on the language presented in the original work.

CSP provides a parallel statement for starting concurrent processes, where each process starts simultaneously and where the parallel statement ends when each process has terminated. The behavior of the processes is described using repetitive and alternative commands based on Dijkstra's Guarded Commands [45], and allows non-determinism as the guarded commands does not need to be mutually exclusive. The processes are only allowed to communicate by updating variables through the communication primitives. Originally CSP did not include any automatic buffered communication, instead, communication in CSP was synchronous and involved a rendezvous between the processes sending and receiving the message, this means that both sending and receiving are blocking operations. Buffering is left out because the rendezvous communication can be used to construct buffered communications by constructing a FIFO process where both left and right sides have synchronous communication. In later versions of CSP, however, explicit channels for message passing are added.

The communication in CSP simply names the process to communicate with and the variable to send or receive. As an example **producer?data;** would receive a value for the variable data from a process named producer while **receiver!data;** would send the content of the variable data to the

process named receiver. The communication only takes place if the two variables have a matching type.

CSP is of special interest as the Spin Model Checker [72] uses a CSP-like language to describe the process network that is analyzed. This language, called Promela (PRocess MEta LAnguage), is briefly presented in Chapter 5 as part of the discussion of how the model checker can be used to extract static schedules from a dataflow program.

Kahn Process Network A simple language for describing parallel programming was presented by Kahn in 1974 [76]. Compared to CSP, the processes in Kahn's model are deterministic, have communication queues with automatic buffering of infinite size, and the processes are designed to run forever. The Kahn Process Network (KPN) achieves these properties by adding some simple commands, for describing how processes relates to each other, to the program description, where the processes are described as an Algol program with the addition of communication primitives.

The communication between processes in KPN is implemented with a blocking receive and a non blocking read. The blocking receive implies that KPN is deterministic, that is, processes cannot check for input, or absence of input [76]. The processes, which are declared similar to a procedure, are run statement by statement until a *wait* primitive is reached, which indicates the read of an input channel. The only way to get input is to use the wait primitive, which simply waits until input is available. With output, on the other hand, the nonblocking *send* primitive is used. With the combination of infinite buffers and nonblocking send operations, sending output is an operations that is always successful.

A KPN is monotonic, which means that more inputs only will result in more outputs and future inputs only concerns future outputs [76, 83]. More formally monotonicity means that $X_1 \sqsubseteq X_2$ implies that $f(X_1) \sqsubseteq f(X_2)$, which means that if a sequence X_1 is a prefix to a sequence X_2, then the resulting sequences of these are also ordered in a corresponding prefix order. In KPN monotonicity is guaranteed by blocking reads, which means that there is only one possible order to read the inputs. A Khan process network is also continuous which means that a process cannot decide to send output only after an infinite number of inputs has been received [76]. These are important properties for the scheduling problem; for models which can have non-monotonic actors, it also implies that the actors are nondeterminate, which makes the scheduling more challenging. As an example, with a non-monotonic actor, a schedule that is correctly generated for a specific input, may be incorrect when more inputs are available.

Dataflow Process Networks Lee et al. presents Dataflow Process Networks (DPN) [83] as a special case of Kahn process networks, where the processes, called *actors*, consist of repeated firings. The DPN model of computation is highly relevant to this thesis as the Cal Actor Language (CAL) is based on DPN. Compared to KPN, DPN adds the principle of firings as proposed by Dennis, which is useful for describing dataflow applications. [85]

The introduction of firings provides a quantum for scheduling calculations that map input tokens to output tokens. The behavior of an actor is then described as pairs of firing rules and firings, where the firing rule defines a precondition for an actor firing, while the firing itself consumes a fixed number of tokens on input streams and produces tokens on the output streams. A *scheduler* then interleaves actor firings according to a set of firing rules which may depend on inputs or the state of the actor. The firing rules are a set of expressions, describing some properties regarding the input sequences and the actor state that are required for an actor firing to be eligible. The actor firings in DPN can to some extent be compared to Dijkstra's Guarded Commands [45] and the firing rules do not need to be mutually exclusive implying that an actor can be non-deterministic. Consequently, non-deterministic actors must be identified, and non-monotonic behavior needs to be identified in order to produce correct scheduling.

DPN is a rather expressive model of computation, which can be used to implement applications that could be implemented with more restricted models of computations. While this means that any dataflow program, or part of a dataflow program, can be implemented with DPN, the implementation in DPN does not automatically provide some of the convenience that more restricted models of computation would provide concerning predictability or static scheduling. Instead, these properties need to be identified by analyzing the actors of the whole program, and, in one way or another, fit the actors in to more restricted models of computation that have the desired properties. In the next section different dataflow models of computation, with different trade-offs between expressiveness and predictability are presented in order to show what kind of properties we want to identify in DPN programs.

The Actor Model In order to avoid unnecessary confusion, a related but still different model, namely the actor model should be briefly mentioned. In dataflow models, a computational node is often called an actor. This is, however, not to be confused with the actor model, which is a separate model of computation.

The actor model was designed as a model of concurrent computation, and was created as a new theory of an inherently concurrent model [37]. Compared to the process networks and data flow networks already discussed in this chapter which have a fixed network topology, in the actor model

the communication topology is dynamically changing. The Actor model allows actors to dynamically create actors, send messages to other actors, and respond to messages received from other actors. The fixed topology of dataflow programs is beneficial for analysis, as the connections between actors are static, the dynamic topology of the actor model, on the other hand, allows more dynamic application models. Which model is better will of course depend on the application that is being implemented, and for many signal processing applications, a static topology is adequate.

The actor model was presented by Hewitt et al. in 1973 [67], and further developments and reading can be found in Hewitt et al. from 1977 [66, 13], Clinger from 1981 [37], Agha from 1986 [6], and many others. To not get off topic too much, the actor model will not further be discussed in this thesis, instead we will move on to discussing more specific dataflow models of computation.

2.2 Dataflow Models of Computation

The behavior of a dataflow program is defined by the Model of Computation (MoC) of the dataflow network, which defines in which order and how the nodes are to be executed. Several MoCs have been presented, each describing a different trade-off between expressiveness and compile-time predictability. The simplest one is Synchronous Data Flow (SDF) which is a statically schedulable MoC [81], while e.g. dataflow process networks are dynamically scheduled, that is, scheduling decisions are taken at run-time [83].

In this context we are particularly interested in compile-time analysis and scheduling of dataflow programs in an SDF like manner but without sacrificing expressiveness of models like DPN. Depending on the MoC which define the operations an actor is allowed to perform, some properties are inherent to more restricted models while for less restrictive models these properties may be more difficult to verify. The properties of a dataflow actor that can be resolved at compile-time depend on either what behavior is allowed by the MoC of the actor or which of the allowed properties are used. In other words, either some properties are implied by the restrictions of the programming model or these properties need to be identified from the implementation. Dataflow models are often used to model and implement signal processing systems which in many cases work on never ending data streams. For this reasons it is important to analyze properties such as, if the program can run using bounded memory, whether or not there exist schedules of finite length that bring the program back to its initial state, and if the programs are guaranteed to be deadlock free. Each of these problems are related to the main problem addressed in this thesis, that is, how to generate a set of static schedules that can describe the different operations

of the dynamic dataflow program.

Numerous computational models have been proposed to find a comfortable trade-off between expressiveness and predictability in the range between fully static models such as SDF and dynamic models such as DPN. There are benefits with requiring the user to specify and *stick to* a specific restricted MoC, by this the programmer is forced to think and specify the intended behavior, but in many cases it is more beneficial to allow a less restricted MoC and let the compiler try to resolve the properties of the program. Dataflow programs can be divided into different MoCs depending on how token rates and firing rules are specified. Several MoCs, which levels of predictability and expressiveness, reside between static dataflow (synchronous data flow) and dynamic dataflow, exist. With static data flow, every scheduling decision is compile-time predictable while with dynamic data flow, scheduling is in general a run-time operation. Typically, each dynamic dataflow program includes portions that are static, and therefore compile-time predictable, and for this reason, dynamic dataflow programs or parts of these, will usually belong to more restricted MoCs with better compile-time predictability.

The work in this thesis focuses on one of the more expressive MoCs namely Dataflow Process Networks (DPN) which is the model used in CAL. It is necessary to present some of the other models as much of the work on analyzing CAL can be compared to fitting a CAL program in to a more restricted MoC in that sense that finding that some language constructs are not used allows more precise analysis of the program. In the next section we will go through a set of MoCs from the fully static models to more dynamic models, and then we will present related approaches on how to analyze and schedule dynamic dataflow programs.

Generally speaking, a node in a dataflow application may correspond to anything between a simple arithmetic operation to a larger subprogram. For any type of platform, a dataflow program requires scheduling, and the larger the number of actors allocated to a specific processor core is, the larger the impact of efficient scheduling becomes. When several actors are mapped to the same processor, the size of nodes should be adapted to fit the target architecture by clustering, and the scheduling of the nodes should be optimized to minimize the scheduling overhead. To achieve an efficient implementation on a multi-core system, the compiler should resolve as much as possible of unspecified decisions while leaving *balancing* for the run-time. Now, we must decide what can be decided at run-time. In theory, all alternative paths of a program can be derived at compile-time, although it may not always be practical. Hoare describes computer programming as an exact science in [68] as "all properties of a program and all the consequences of executing it in any given environment can, in principle, be found out from the text of the program itself by means of purely deductive reasoning". In practice this means that, if the program is known, we can calculate the outputs

for every possible input. If we look strictly at which parts or statements of a program are executed, we can deduce a set of sequences of statements that are sufficient to process every possible input of the program. To create efficient dataflow programs, different programming models have been proposed that guarantee different properties, by construction of a program according to that MoC. Many of these provide programs that allow static scheduling, unbounded buffers, or deadlock freeness.

Dynamic Data Flow A dataflow program where the computations are described with a set of firing rules, depending on variables or data tokens that are evaluated at run-time, belongs to a MoC called Dynamic Data Flow (DDF). The concept of DDF is loosely defined in literature, and seems to in many cases describe a model that does not correspond to any of the more restricted MoCs. In this text the simple definition of DDF used in [122] will be used to describe applications where no scheduling information is given for the program. In a DDF program, an actor has a set of firing rules which can be any Boolean expression and the tokens rates can be undefined or be defined as a range [122]. A DDF program can also be non-deterministic by allowing more than one firing rules to be enabled at once. According to this definition, DPN can also be considered to belong to DDF.

With DDF scheduling is a run-time operation; some or all scheduling decisions can however be predicted at compile-time by either describing the program with a more restricted MoC or by analyzing the program to find if parts of it can be described with more restricted MoCs. This operation can be seen as a classification of one MoC to a more restricted MoC, and can be described as identifying if a program implemented with a less restrictive MoC fulfills the requirements of a more restrictive MoC.

Synchronous Data Flow One approach to solve the scheduling problem is to use a restricted MoC which enables static scheduling. Many signal processing algorithms, such as a filter, can be described as a dataflow program where each node has constant data rates. Such programs can be modeled as Synchronous Dataflow (SDF) which can be statically scheduled and are optimal as no scheduling decisions are needed at run-time. The static scheduling of SDF programs is presented by Lee in [81], and the conditions for the existence of a valid static schedule has been extended for many of the other MoCs we will review as well.

The number of times the nodes in an SDF graph must be triggered in order to form a periodic sequential schedule is found by solving the balance equation $\Gamma q = 0$, where the (i, j)th element in Γ represents the number of tokens node j will produce on arc i; in case the node consumes tokens the entry will be a negative number. If the equation has a non-trivial solution,

that is, a non-zero positive solution, there exist a schedule that will take the network back to it initial state. A necessary condition for the existence of such a schedule is that $rank[\Gamma] = s - 1$, where s is the number of nodes [82].

The program is said to be consistent if the consumption and production rates of each edge match in the long run [79]. Consistency does not mean that it is possible to construct a periodic admissible schedule. As an example, every cycle in the dataflow graph with an initially empty buffer is deadlocked. By inserting the appropriate initial tokens (delay tokens) $b(n) = b(0) + n\Gamma q$ it will be possible to create a schedule that runs one period without consuming tokens from any empty FIFOs. [82, 81]

For an SDF graph, some important properties can be proved. If the token rates are consistent, the program can be run, also in the long run, with static FIFO sizes (bounded memory). This is an important property for static scheduling as inconsistent token rates can lead to accumulation on tokens which imply either unbounded memory requirements or deadlock.

Composition of two adjacent nodes is not always possible without introducing deadlock. Pino et al. [105] presents an SDF composition theorem for testing if a composition (or clustering) is valid and does not introduce deadlock. Generally speaking, an SDF graph does not deadlock if and only if it has an acyclic precedence graph [105]. The composition theorem includes four criteria that prevent cycles in the precedence graph of the composed actors.

Not every application can be modeled as SDF and for this reason there exist several *almost SDF MoCs*. These typically have a deterministic behavior but are somewhat more complex than SDF and the MoCs are designed to allow some flexibility to describe algorithms with some specific behavior.

Cyclo-Static Dataflow One extension of SDF is Cyclo-Static Dataflow (CSDF) which allows the dataflow network to have different data-rates for different firings as long as these are periodic. This is useful for applications with cyclically changing behavior, for example, one node in the dataflow network have different data-rate every other time it is executed. The resulting application can still be scheduled statically at compile-time by using an extended version of the methods for scheduling SDF. [23]

At first glance, CSDF may seem more like a convenient representation for some SDF programs as an CSDF actor can be described as SDF if the cycles are merged. However, the CSDF representation offers some benefits over the SDF representation as the SDF model may introduce deadlock in a dataflow graph where it could be avoided with the functionally equivalent CSDF actor. The CSDF MoC also enables reduced buffering requirements and network delay by allowing actors to fire *partial* firings which, if fired at once, would correspond to an SDF actor with higher token rates.

Token Flow Model Another type of extension needed in many programs is the ability to describe conditional execution or a choice between several alternatives. A basic model for this, called the token flow model, is presented by Buck in [31]. In this model special nodes, which are called Switch and Select, has an special input which carry a Boolean value deciding which input/output is active at that specific time. An extension of the token flow model allowing integer values instead of only Boolean values is presented in [30], the integer value can be used to either specify the number of tokens that are produced or consumed, or to enable/disable an arc depending on the value of the token.

To determine if a graph is consistent the balance equation is required to have a nontrivial solution. When the dataflow graph has dynamic actors, the topology matrix will have entries with p_i corresponding to the proportions of tokens with a special value. The graph is called strongly consistent if the balance equation has a nontrivial solution for any value of p_i and weakly consistent if the solutions only exist for some values. A graph with strong consistency will have bounded buffer requirements while a graph with weak consistency only for the right proportions of the control values.

For both the Boolean and integer dataflow models, it is possible to find if the graph is consistent and can be scheduled with bounded memory. This also means that a set of static schedules which can be chosen based on the control values, can be derived. If a dataflow graph is strongly consistent depends in its topology and how the special control actors are used. A slightly different approach is to restrict the usage of control actors such that a design always will have the desired properties.

Well-Behaved Dataflow Gao et al [58] proposed *well-behaved dataflow*, where the dataflow graph is built from actors corresponding to the actors in the *token flow model*, but allowing only structured use of Switch and Merge (Select) actors. Well-behaved graphs only allows the dynamic actors in the *conditional schema* and the *loop schema* which enables compile-time predictability of the storage requirement [58].

A MoC which somewhat combines the benefits of both CSDF and BDF is the Cyclo-dynamic Dataflow model.

Cyclo-dynamic Dataflow Cyclo-dynamic dataflow model (CDDF) extends cyclo-static dataflow [122] such that it can cover all BDF and CSDF graphs. The CDDF model extends the CSDF model and keeps the properties related to analyzability and compile-time scheduling, but introduces data dependent choices similar to those of Boolean dataflow.

What is interesting with CDDF is that it gives the designer the opportunity to express additional scheduling related information that is not possible

in the other models. A token rate of a port of an actor can, for example, be described as a sequence of rates, like in CSDF, or as a function of an input value. In this way, the dynamic behavior of these actors is explicitly described and the model can be said to tie the output of one actor to the corresponding action in the other actor when these expressions are combined. For dynamic dataflow models, compile-time scheduling often requires this information to be derived from the implementation in order to predict the behavior of the model.

Some MoCs handle the control or dynamic behavior on a higher level.

Parameterized Synchronous Dataflow The Parameterized Synchronous Dataflow (PSDF) is presented as a meta-modeling technique in [20]. PSDF allows parameterization of sub-systems where parameters can control functional behavior and also token flow behavior of the dataflow graph. A parameterized graph behaves like a graph in the underlying dataflow model, during each of its invocations, but can assume different configurations across invocations [20]. The PSDF model assumes SDF as the underlying model, however, any underlying dataflow model which has a notion of a graph iteration can be used. For example, the concept of periodic schedules or iterations exists in SDF, CSDF, Multi Dimensional SDF [80] etc.

A PSDF specification Φ consist of three graphs, namely, the *init graph* Φ_i, the *subinit graph* Φ_s, and the *body graph* Φ_b. The body graph represents the dataflow program while the init and subinit graphs are used to configure the parameters of the body graph.

With the PSDF model, a subsystem, say the texture coding part of a video decoder, could have two behaviors, for decoding macro blocks (MB) with or without intra prediction. For each MB processed, the texture coding subsystem is configured to process the next block type. Further, the parameterization could be implemented by a data input from another subsystem, which is read and used to reconfigure the dataflow graph between the invocations. This kind of scheduling resembles the type of scheduling which is the goal of the methods presented in this thesis when the *subinit graph* Φ_s is seen as the choice of schedule and the *body graph* Φ_b in the CAL program has different token rates as a result of a different schedule being fired.

A MoC with a somewhat similar reconfigurability, but where the control part is more integrated in the MoC, is the Scenario-aware Dataflow model.

Scenario-Aware Data Flow A dataflow MoC which also handles control tokens or dynamic behavior is the *Scenario-Aware Data Flow* (SADF) [116, 113]. Here, the dynamic behavior is viewed as a set of *scenarios* occurring in some possible order, modeling the behavior of each scenario as an SDF graph while the dynamic behavior is modeled as a set of possible scenarios [113].

An SADF graph can be viewed as an SDF graph where some of the token rates are indicated as a variable instead of a constant as in SDF. The actors in an SADF graph are of two types: *kernels* which are the data processing part of the program, and *detectors* which models the control part of the application. When a detector fires, it sends control tokens indicating the detected scenario and fixing the values for the parameterized tokens rates of the receiving kernels [116]. SADF enables a pipelined execution of scenarios as the control tokens are part of the model. The SADF model has been designed to allow efficient performance analysis, however, only the MoC is relevant for this thesis.

A restricted form of the SADF model adds a finite state machine (FSM) to represent the possible order in which the scenarios can occur [113]. This model, called an *FSM-based SADF graph*, resembles to some extent the type of dataflow scheduler that is generated from dynamic dataflow applications with the approach presented in this thesis.

Points of Interest The different MoC presented in this section provides different restrictions or descriptions of relevant information that makes the dataflow program analyzable. In the context of this thesis, this is interesting as scheduling of a CAL program is about identifying properties, similar to these presented in the various MoCs. On a high level, the interesting information relates to how the token rates of an actor firing can be predicted, and how the tokens that carry information that will affect scheduling is described in the MoCs that allow data dependent firing rules. In the last section of this chapter, the CAL actor language is presented with focus on the different constructs that are relevant for scheduling.

2.3 The Cal Actor Language

The Cal Actor Language (CAL) is presented by Eker and Janneck in [46] as an actor language which is part of the Ptolemy project [29, 1], at the UC at Berkeley. CAL is a very expressive language for describing dataflow actors, and can implement the DPN model, which in practice means that any dataflow program can be implemented with CAL.

Since the introduction of the language, CAL has gained interest also within other contexts than Ptolemy, applications within signal processing, cryptography, and multimedia, has been implemented using the CAL language. The perhaps greatest achievement so far is that a subset of CAL named RVC-CAL, has been standardized by ISO/IEC MPEG [2, 93], for the Reconfigurable Video Coding (RVC) initiative. In MPEG RVC a video decoder is described by a network configuration of RVC-CAL actors instantiated from a standardized library. This decoder description is associated

```
actor actorName() uint(size=8) Input ==> uint(size=16) Output :

 uint(size=8) previous := 0;

  firstAction: action Input:[ x ] ==>
  guard
    x > 0
  end

  secondAction: action Input:[ x ] ==>
  do
    previous := 0;
  end

  thirdAction: action Input:[ x ] ==> Output:[ x * previous ]
  do
    previous := x;
  end

  schedule fsm s_start :
    s_start ( firstAction ) --> s_work;
    s_start ( secondAction ) --> s_start;
    s_work ( thirdAction) --> s_start;
  end

  priority
    firstAction > secondAction;
  end
end
```

Figure 2.1: Example of a CAL actor.

to compressed video content, thus providing both the compressed video as well as all the information for decoding it.

Because CAL is general enough to implement applications belonging to the DPN MoC, a CAL program is in the general case not statically schedulable. Instead, scheduling is a run-time operation, where firing rules are evaluated and an actor fires whenever a firing rule evaluates to true. The actors only communicate using FIFO channels, and each actor can fire concurrently with the other actors as long as there is a firing rule of which the condition is fulfilled. Each actor requires some scheduling, that is, deciding on what to fire and in which order. For the example program in Figure 2.1, the behavior is described as an FSM with two states, where the state named *s_start* enables the two actions *firstAction* and *secondAction*, while the state named *s_work* enables the action *thirdAction*. The scheduling of the actor is then about determining which of the actions enabled in the current FSM state of the actor are eligible for firing by evaluating the

guard expressions of the actions and the requirements regarding availability of tokens on input FIFOs and space on output FIFOs. On the program (network) level, scheduling is concerned with finding an actor which can fire, which in practice means that it has at least one action which is eligible for firing. This scheduling becomes a practical problem when several actors run concurrently on a single processor, and hence the scheduling needs to be efficient.

2.3.1 Relevant Language Constructs

A CAL actor executes by firing actions. An action describes the relation between input and output ports, and when an action fires, it may consume tokens from input ports, produce tokens to output ports, and modify the state of the actor. The token rates of the actions are fixed, and are defined in the program description, as an example, consider the following action which reads two tokens from input A and writes one token to port C.

```
actor actorName() uint A ==> uint C :
  add: action A:[a,b] ==> C:[c]
  do
    c:=a+b;
  end
end
```

If this action is the only action of the actor, the actor is obviously SDF as the token rate is static for each firing; this is true for any action. The dynamic behavior of a CAL actor appears from the interaction and scheduling of the various actions the actor contains. This scheduling can depend on 1) the state variables of the actor, 2) the action scheduler, and 3) input values.

Firing rules The *state variables* are, as the name hints, variables that keep their value between action firings, and are therefore part of the actor's state. A state variable is used to store information that is needed between action firings, whether it is data or control. The variable *previous* in the actor in Figure 2.1 is an example of a state variable that is used for the actual data processing. A state variable may also keep control information; this simply means that the variable is used in the firing rules, called guard expressions, of an action; as shown in the following example.

```
actor actorName() uint Input ==> uint Output :
  uint count := 0;

  action Input:[x] ==> Output:[x]
  guard
    count < 64
  do ... end
end
```

The other part of the actor state is described by the action scheduler. This construct, if present, is (typically) a Finite State Machine (FSM) which restricts when action may fire. The FSM describes a set of states and the transitions between these states correspond to actions; only actions corresponding to outgoing transitions of a specific state may fire in that state. The FSM, as such, already describes the order in which actions may fire and also shows the states where a scheduling decision based on a guard is needed, as such states have more than one outgoing transitions. While the FSM also could be implemented using state variables, from a scheduling point of view, the FSM is more practical as it describes the intention of the programmer. An example of the FSM scheduler can be seen in Figure 2.1.

Except for the actor state, the scheduling of actions may depend on the inputs of an actor. An action guard may directly refer to a value on an input port, which means that the token is read but only removed from the queue if the guard evaluates to true and the action fires. This is called a *peek* and can be used to implement behavior where the appropriate action is schedules based on the value of an input token. Consider the following actions as an example.

```
actor actorName() uint A, uint B ==> uint C :
  divide: action A:[ x ], B:[ y ] ==> C:[ x/y ]
  guard
    y != 0
  end

  skip: action A:[ x ], B:[ y ] ==>
  guard
    y = 0
  end
end
```

This simple actor, divides a stream of values by two, but avoids division by zero, by skipping the pairs where the denominator has a zero value. A guard may also depend on an input value that is first read to a state variable, technically, the guard simply uses the value of the state variable, but from a scheduling point of view, the value of the state variable is now much more difficult to predict.

The guard expression is the explicit part of the firing rules. For an action to fire, the guard expression must be evaluated to true, but there must also be a sufficient number of input tokens available, and in a practical implementation with limited buffer sizes, enough space on the output queues. As the actions have fixed token rates, it is trivial to extend the firing rules with the check for available inputs and output buffer space.

Special properties So far, the discussion has concerned the basic constructs and firing rules of a CAL actor. Depending on how a CAL actor is

constructed from these, it will have different properties, and may conform to more restricted MoCs than DPN. These can also be used to construct actors with more interesting, but also more difficult to handle, properties that are a result of firing rules that are nondeterminate or not mutually exclusive.

A nondeterministic actor can simply be created by having two actions that can be enabled simultaneously, which means that correct behavior is preserved if either of the actions is fired. Nondeterminism is useful for some actors when the programmer does not want to over specify the operation; consider the following actor as an example.

```
actor actorName() uint A, uint B ==> uint C :
  one: action A:[ x ] ==> C:[ x ] end

  theOther: action B:[ x ] ==> C:[ x ]  end
end
```

In this actor, if tokens are available on both input channels, the choice between the actions *one* or *theOther* is not present in the actor description. This kind of an actor has both desired and non wanted properties; the actor is flexible as it can respond to input on any of the inputs at any point in time, on the other hand, if the input queues are constantly filled with more data, the actor may decide to only server one of the inputs and never choose the other one. Nondeterminism can be identified by comparing the guards of actions that can fire in the same state, if the actions have mutually exclusive guards, the actor is deterministic.

Another related property is when the firing rules depend on a volatile property which means that which action will fire depends on the point in time when the firing rules are evaluated. Time-dependent behavior appears when a lower priority action may fire when the other action does not have sufficient inputs. This happens when the guards of two actions are not mutually exclusive but the choice of action is specified by priorities. The following example shows an actor where one action fires as long as there is input while the other becomes enabled when the input is empty.

```
actor actorName() uint A, uint B ==> uint C :
  one: action A:[ x ] ==> C:[ x ] end

  two: action  ==> C:[ 0 ]  end

  priority
    one > two;
  end
end
```

It is important to identify time-dependent actors when analyzing scheduling properties of actors; a method for identifying time-dependent actors is presented by Wipliez et al. in [124]. Time-dependent actors are related

to the property of monotonicity; if an actor produces a specific output sequence for one input, adding more input tokens does not simply mean that the output will be appended with more tokens.

The properties presented, are relevant for scheduling actors and actor partitions. For a scheduler to be complete, it must take into account data dependent firing rules, nondeterministic actors, and time-dependent behavior. A scheduling model must either verify that these properties are not present in the model or that these are preserved in the resulting scheduler such that the new model accepts all the input sequences the original program accepted.

2.3.2 Scheduling and Code Generation

The code generation from CAL is concerned with 1) translating CAL actors into a language such as C, which then can be compiled into native code for the target platform, and 2) generating the communication network and a run-time scheduler. The code generation process is presented by Wipliez et al. in [125], where a code generator named is Cal2C is presented. There is a number of CAL compilers and tools available [22, 110] and the one used in this work is the Open RVC-CAL Compiler (Orcc) which incorporates Cal2C but also provides the infrastructure for building the tools needed for this study. There are in principle three aspects that are relevant for generating efficient code. First, the actual translation of actors to a low-level language, second, how efficiently the run-time scheduler finds eligible actions, and third, how the actors can be simplified by applying static scheduling on individual actors of actor partitions. This thesis focuses on the third aspect, and tries to optimize actors by composing actors into suitable size for the target platform and by simplifying the action selection process of these actors.

There is a number of approaches for scheduling CAL programs in literature that should be mentioned. Although we call a program dynamic, there always exist parts of the program which can be scheduled statically which means that quasi-static or piecewise static approaches, where only necessary scheduling decisions are taken at run-time while most of the scheduling is performed at compile time, can be used to increase the performance of the application. Generally speaking the approaches can be divided into two categories, the first has, or acquires, some information about actors which is used to decide in which order actors should be fired, while the other looks at how an action in one of the actors enables other actions across the dataflow graph. Approaches for scheduling CAL programs will be presented in Chapters 4 and 5.

Chapter 3

The Scheduling Problem

Every dynamic dataflow program can be scheduled with a finite set of static schedules. On the one extreme, a static schedule is a single action and a scheduling decision is made between each action firing; on the other extreme, a static schedule is identified for each combination of program state and input sequence of finite length. Somewhere between these two ridiculous extremes, we can find more practical trade-offs that will affect different properties of the program such as the number of conditional branches a program will have or the cache behavior of that program.

The scheduling of a dataflow program is not a trivial task when performance, measured in whatever sense, is the goal. As will be further discussed in Chapter 6 (and shown by the results in Paper III), it is not always the configuration with the least run-time scheduling decisions, and the largest actor compositions, that will give the best results, but instead the scheduling should be tuned to fit the chosen hardware architecture. As a result, scheduling should be performed in smaller steps where the first steps are platform independent and are used to transform the program in to a representation that can be used to produce schedules according to the properties of the platform. In practice, this means that a program or part of a program first should be translated into a representation similar to a partial order from which actual schedules can be generated.

Consequently, scheduling can be viewed as the following operations. First conditions that are definitely compile-time are identified; these are guard evaluations that can be decided from within the actor at compile-time. Next, inter action dependencies are analyzed and based on this, the conditions are found to be either static or dynamic depending on how actors are composed. Last, the actors are composed according to these dependencies such that the platform decides the size of the compositions.

The scheduling problem can in simple words be described as, resolving how parts of a dynamic dataflow program can be identified to belong to

MoCs with more predictable properties. This can be achieved either by deciding a MoC per actor and then using existing scheduling approaches to schedule the actors, or by fitting a dataflow graph partition into a MoC.

3.1 Specifying Dynamism

The scheduling decisions of a program must be decided to be either run-time or compile-time. For most decisions this is obvious, but for some, the choice is a trade-off between the complexity of the guards that need to be evaluated to choose a static schedule and the number of action firings that can be scheduled based on the guard. The topic of this section is to describe the difference between these types of conditions and to describe what makes a dataflow program dynamic.

From the point of view of a CAL actor, dynamic behavior is driven by the input sequences. A dataflow program may allow more than one firing sequence to process a single input sequence, however, this is not considered to be dynamic behavior, in this context, but instead it is simply considered a different interleaving of actions where the exact order does not matter for the result. The dynamism of interest is instead when the value of, or the order in which, the input arrives makes the program act differently. This is illustrated in the two actors in Figure 3.1 where the first actor shows a non-deterministic behavior where the value from either of the input ports X and Y is copied to the output port Z, and the second actor consumes from input ports X and Y, but only produces output on the port Z when the input on X is a non-zero value.

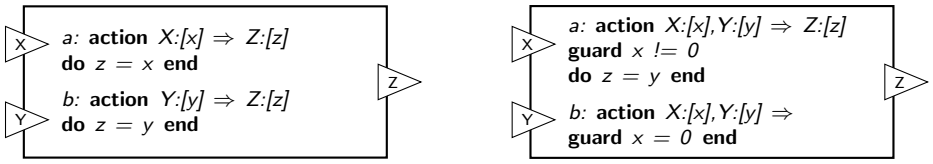

Figure 3.1: Two actors with dynamic behavior.

These two actors represent two different types of dynamic behavior. For the first one, firing any of the two actions results in correct behavior as long as input is available. However, if we create a schedule, for example, where actions a and b are fired every other time, the token rates of the program may not be consistent and consequently, one of the buffers may grow infinitely and in practice cause deadlock. For the other actor, such a schedule cannot be produced as each action firing depends in the next input value on X. For neither of these actors, is it possible to find static schedules or action sequences with more than one action (except if we cover every combination of input sequences).

One more type of dynamic behavior is when an input value or variable which is used for scheduling may obtain many values and each of the values results in a slightly different schedule. A practical example of this is a run-length decoding action which reads an input and the read value decides how many output tokens will be generated by firing an action that many times.

When one of these dynamic actors is put in a context, the dynamism may be resolved by the surrounding actors. For this reason, scheduling can often be simplified by analyzing actor partitions instead on single actors. In Figure 3.2, one of the actors which required dynamic scheduling in the previous example, is connected to an actor which makes it possible to remove the run-time scheduling decision. The first actor in this small dataflow network repeats the input sequence from input port I to the output port K and produces the sequence $0, 1, 2, 3, 4, 0, 1..$ on output port J. The combined behavior of these two actors, is that, every fifth token is removed from the stream, and a static periodic schedule, such as "s,b,s,a,s,a,s,a,s,a,t", can be produced for the two actors.

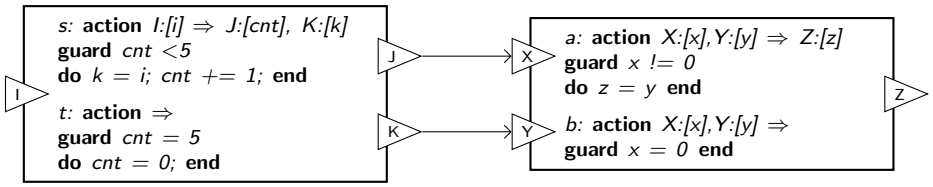

Figure 3.2: One of the dynamic actors put in a predictable context.

Imagine for a while that the actors in Figure 3.2 are connected differently; port J is connected to port Y and port K is connected to port X. Now the choice between actions a and b is not anymore predictable from the other actor as the value enters from outside the dataflow network. Then again, if we replace the second actor with the non-deterministic actor from Figure 3.1, it will again be possible to create a static schedule because the tokens rates on X and Y are known.

From this discussion it should become evident that two aspects need to be modeled in order to decide if a scheduling decision can be performed at compile-time. The first property is related to the propagation of control tokens. In the example in Figure 3.2, the control token is both created and used within the two actors in a predictable fashion, which makes joint scheduling of these actors attractive. The second property relates to consistency of the dataflow graph. When there are actors with, so to say, flexible token rates in the dataflow graph, static scheduling is possible only if this flexibility is only available inside the dataflow graph but not available on external ports.

The following two sections describe how these properties can be ana-

lyzed by constructing simple graphs which highlight where joint scheduling is possible and can be used to perform composition of actors.

3.2 Scheduling Based on Partition Input

"The choice of a static schedule for an actor composition can be made based on the input sequence and on the state of the partition."

Perhaps the most obvious type of dynamic behavior is when the behavior of a program depends on the type of data that is fed to the program. The program reacts to the data and there is clearly a need for run-time decisions, choosing the appropriate operations based on the incoming data. The type of scheduling targeted in this work is a partition of actors, where a static schedule can be chosen by inspecting the incoming data sequence, and then schedules this sequence through the partition.

In a CAL program this means that there are guards that depend on the value of an input token (or the value from a native function call). From the actor itself, it cannot be known whether the guard will evaluate to true or false unless the value of the input token is given. However, the question is whether or not there are redundant guards in the partition, that is, more than one guard check the same property and one guard can be predicted from another. Such redundancy may reside inside an actor but more often it can be found between actors as was shown in the examples above. To find out if guards checking a control value are redundant, or if the guards checking the incoming data are strong enough to schedule the partition, it is necessary to analyze how the control information is distributed in the partition (see Figure 3.3).

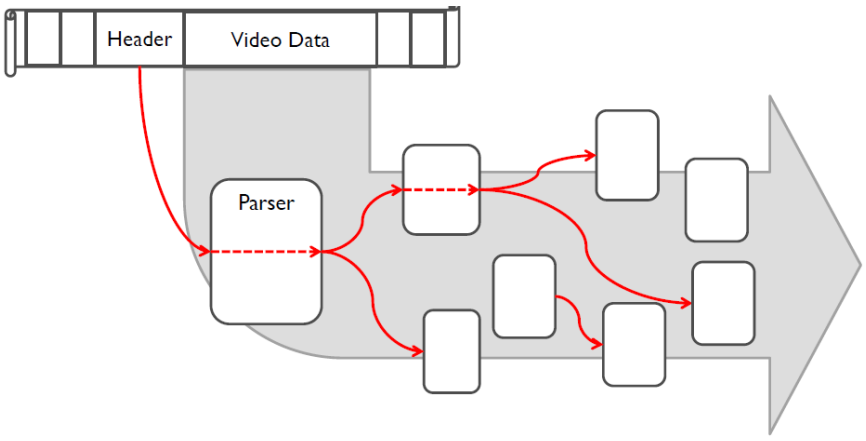

Figure 3.3: Control values typically appear from the input bit-stream and are then distributed in the program and finally ends up in guard expressions.

Scheduling a partition of actors based on the input sequences to the partition requires some properties to hold regarding firing rules and how the information propagates in the partition. Before discussing these properties, some definitions are needed.

Definition 3.1. *Variable Dependency Graph. The internal behavior of the actors is described as a directed graph* $G^R = (V, R)$ *where* V *is the set of variables and the edges are described by the binary relation* $R \subseteq V \times V$ *such that* $(v_1, v_2) \in R$ *indicates that 1)* v_1 *is assigned a value based on variable* v_2*, or 2)* v_1 *is associated to an input port which is connected to an output port associated to* v_2*.*

Definition 3.2. *Control Token Path. A control token is a value on a FIFO that eventually ends up in a guard expression. This means that, for each variable directly used in a guard, each node reachable in* G^R *is part of the control token path.*

Definition 3.3. *Guard Strength. A set of guards are sufficient (strong enough) for scheduling a partition when these describe every scenario of the control token paths of the partition.*

3.2.1 Choices and Redundant Choices

"Efficient scheduling means that once a property is checked when data enters the partition, it should not be checked again inside the same partition."

To understand why redundancy can be expected in a CAL program, it is necessary to pay some attention to why dynamic behavior typically is introduced in a program. There are mainly two sources of dynamic behavior. The main driving factor is the input-stream which, depending on its contents, requires the program to act differently. The second factor is the implementation choices, that is, how the algorithms have been implemented; this also includes how the information from the input-stream is described and distributed in the dataflow network.

To begin with, the input-stream, as such, defines how much of the scheduling must be performed at run-time. If there is a part of the input bit-stream that can hold one of several types of content, the application must also have the corresponding check to choose the appropriate functionality for the content at run-time. A stream parser, most usually, makes this check but also the other actors that depend on this check typically receives this information as a control value on which they make a check. As an example consider a video decoder application where a parser reads the input stream and produces blocks which are distributed to the different coding tools; see Figure 3.3. A block can be represented by motion vectors (MV) and the 64 coefficients representing the block, but, it is also necessary to

distribute information about how the blocks should be decoded. In a CAL program, the only possibility to distribute this information is to send it as a tokens on the communication channels to each of the actors that need the information, and each actor has separate guards for deciding what needs to be done.

When scheduling depends on the values of inputs, it is typically one the three situations: 1) one of a few control values from the bit-stream, 2) additional scheduling introduced by the programmer, or 3) an actual data dependency. In the first case, the control token is the implementation of distributing information about the bit-stream to different parts of the program. The scheduling decisions could be made already while parsing the stream and the control token is the means to share this information, however, by default, every receiver of this control token must check its value in order to fire the correct actions. In order to schedule network partitions statically, these redundant checks needs to be identified.

In some cases, a guard checks the value of actual data tokens. In such a case, this is a run-time check as it is part of the data processing. One example of such a case is that an actor must fire different actions when a value which is a function of inputs grows larger than a threshold. It is, in general, difficult to automatically make a distinction between a real control token and a data token; one distinction is the number of different values it can take. In the next section we will show how control values are processed in actions and whether these should be regarded as data or control, this analysis is necessary for partitioning a CAL program into scheduling units such that redundant scheduling is removed without altering the functionality of the program.

3.2.2 Control Token Paths and Guard Strength

"A guard of a partition must uniquely describe all the scheduling decisions related to the input sequence."

Control values flow through a dataflow network and finally end up in a guard expression. If we turn this around, a control path can be described as a dependency starting at a guard expression and continuing backwards trough the dataflow graph towards the input stream. The control values pass between actors according to the network of FIFO connections and passes trough actors as dependencies between ports potentially through state variables. By describing the dependencies through actors, each guard can be described as a graph describing how values flow through the network and end up in that guard. This can then be taken one step further to compare guard expressions on the path in order to identify redundant guard expressions.

The type of composition that is desired is shown in Figure 3.4, which shows two actors, where a control token is passed on the FIFO called C, and

where firing action a results in that action c will fire, while action b results in action d being fired. The composition merges the actions into the two actions a, c and b, d, which are scheduled according to the scheduler that originally scheduled actions a and b. In order to do this, it is necessary to show that the guards that are chosen for the scheduling are strong enough to correctly handle every possible input. The goal is to choose one actor and use the input dependent guards of this to schedule the composition; for this actor it must be shown that the guards, so to say, cover the guards of the other actors.

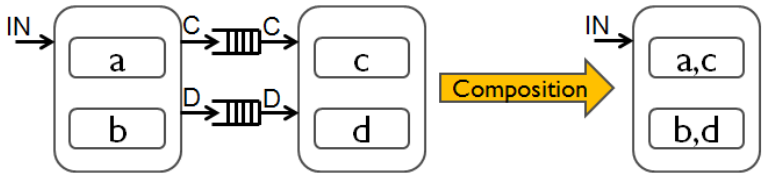

Figure 3.4: The desired kind of composition.

The actors can be viewed as two sets of actions $A = \{a, b\}$ and $B = \{c, d\}$, and a control token belongs to a set of possible control values $t \in T$. The relation between actions and control tokens can be described as two relations, where $O_A \subseteq A \times T$ describes the tokens that can be generated from each of the actions in actor A, and $I_B \subseteq T \times B$ describes the tokens that are accepted by the respective actions in actor B. These relations can be derived from the implementation of the actions and can in some cases be further restricted by analyzing which values may enter actor A in case inputs are used to generate the control output.

The composition with respect to the control values is allowed if a specific action firing in actor A results in a specific action becoming enabled in actor B. The relation between the actions in the two actors can be described as the composition of the relations to control tokens as $f = O_A \circ I_B$, or more practically as the relation between the actions such that

$$f = \{(x, y) \in A \times B \mid \exists t : (x, t) \in O_A \wedge (t, y) \in I_B \wedge t \in T\}$$

If the composition $f = O_A \circ I_B$ is functional, then the composition of the actors can be performed using the guards of actor A. What this means is simply that firing a specific action in A results in a control token that is accepted by a specific action in B. This can also be described as,

$$\forall (x, t_1), (x, t_2) \in O_A : [(t_1, b), (t_2, c) \in I_B \Rightarrow b = c] \tag{3.1}$$

This means that if an action produces two different control tokens, which are accepted by actions b and c, then for f to be functional, b and c must be the same action. In Figure 3.5, this means that every path from one of the

actions in A ends up in the same action in B. To be able to reason about whether this property holds for an actor partition, a model that describes how the control tokens are generated is needed.

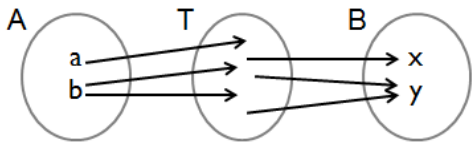

Figure 3.5: The relations between actions and control tokens.

3.2.3 Variable Dependency Graph

In order to describe how a control value is created, a graph representation of the dependencies between variables is generated, where variables and ports are represented as vertices and edges represent dependencies. For each instruction in an actor, where a variable is assigned a value from another variable, an edge is added to the graph between the corresponding nodes (a more detailed description in Paper 4). The individual graphs of the actors can be composed by adding edges corresponding to the FIFO connections of the CAL program.

By choosing the set of variables directly accessed by a specific guard expression form the graph and extracting the part of the graph which is reachable from these variables, a graph describing the exact generation of the control token that ends up in that guard is created. This graph is, however, rather complicated, as it includes every instruction that is executed when the action computes the control token. The graph is, for this reason, simplified to describe the relations in such a simple manner that it is easy to draw some conclusions from it.

A new simplified graph is generated, where, for each actor, the ports sending control tokens and the guards, are nodes with an edge to each state variable and port which is reachable from it in the original graph. Other variables that are not reached from these and local variables are removed from the graph. For the reachable variables, the only edges that are added to these are those to input ports or self loops. This is illustrated in Figure 3.6.

From the simplified graph the dependencies and the complexity of the control token is easily identifiable. The types of control token generation in an actor can informally be divided in to three groups with different levels of complexity:

Galvanic Isolation The simplest form of control token is called *galvanic isolation* to highlight that there is no connection between the output port

sending the control token to any input port, instead the connection is implicit in the sense that the input may decide which action is fired, but the input values are not directly used to create the control token. For a specific state of an actor, this type of control token is deterministic from the actor itself and the guards are always sufficient to schedule the composition. This is true because the control token is only used to schedule that specific actor and the guards of the actor are obviously strong enough for this purpose. Any potential control token generated from this actor only depends on input or the guards in the sense that it is generated as a result of which actions are fired.

A composition of two actors with this kind of a control token connection may include removing guards which depend on state variables in either of the actors. The control token may thus depend on control variables but those control variables may not depend on inputs. The value of this variable must

```
actor example () IN ==> OUT :
  var int count := 0

  a: action ==>
  guard count <= 0
  do
    count:=100;
  end

  b: action IN:[i] ==> OUT:[o]
  guard count > 0
  var int step;
  do
    step:=i;
    count:=count-step;
    o:=count+step;
  end
end
```

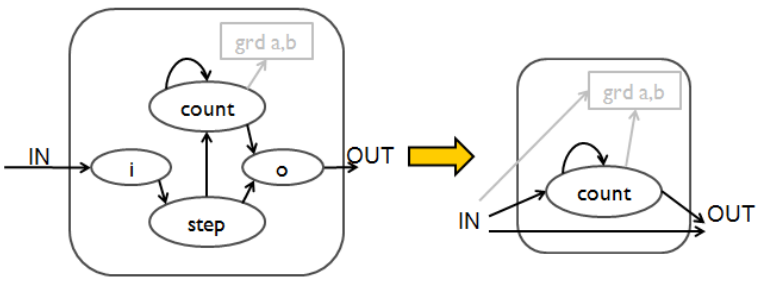

Figure 3.6: Dependency graph simplification. Temporary variables such as *step*, *i*, and *o* are removed, and the edges of the graph are simplified to highlight the dependencies of a control value.

then be used to describe the state of the partition when the scheduling is performed.

Actors with this kind of control token dependency, can from the point of view of control tokens, always be composed. The obvious reason for this is that the control tokens entering a partition are directly used in guard expressions after which they are never used for scheduling again. Any other guard is only indirectly depending on the control token and may become enable as a result of which action was fired. For scheduling, this is a good situation, as the dependencies inside a partition are between action firings and not input values.

Shared Property A slightly more complicated type of control token propagation is when there is an actual connection between a control token entering the actor and the control token leaving the actor, but the generation of the control token is instant and completely done within an action firing. The control token may depend on input ports, but not on variables depending on input ports. This restriction has two benefits, first, the relation between the input value, the guard, and the output value, is simple. Second, the output control token only depends on the inputs read in that firing but not on inputs from previous firings.

The implication of this type of control tokes is that the guards checking the inputs of a composition must be proved to cover all the properties that affect the actors of the composition. Fortunately, the restrictions regarding how the control tokens are generated makes the verification simple. As there is no history of control tokens used, there is no need to verify properties of input sequences but only on individual control tokens. This fits well with formula (3.1) which can be verified for every pair of possible input values; this will be more closely discussed in Chapter 5.

Shared Information Path The third type of control information path does not have any of the restriction of the two previous types. In the general case, it is difficult to analyze this kind of control tokens, as the value of the token may be a result of the history of inputs. By allowing the control path to include state variables, the propagation of control tokens through an actor is not necessarily purely the firing on an action, but the value of an input may reside in the actor between actor firings and be used at a later point to produce the output control token.

The main problem with this, is, that it is not anymore enough to analyze single input tokens, but instead, sequences of inputs may be required. Composition of actors with this type of control token propagation is not performed, in the general case, as the control is more close to being part of the computations than simply propagating information. In some cases,

however, it is possible to transform the actor to correspond to the previous type, e.g. by merging actions that always fire in sequence and replacing state variables by local variables. In other cases, when a state variable with an *unpredictable* behavior is used to produce a control value, the state variable itself could be viewed as an input port, which then would move the behavior of the variable outside the analysis.

While some automation of the verification of the guard strengths regarding control tokens and transformations of the actors to allow this verification should be possible, it is outside the scope of this thesis. Instead this thesis focuses on showing what is needed to obtain correct models, and how this can be used to allow efficient composition of actors.

3.2.4 A Few Complex or Many Simple Guards?

"A few more simple guards may be less of an effort than a few complex guards."

The process of building the scheduling models is concerned with finding a trade-off between, on the one hand, complex guards and a large number of guards to choose from but large partitions with long static sequences of action firings, and on the other hand, simple guards and scheduler but shorter schedules and smaller partitions. The basic idea in the approach we have used in this work, is to avoid making the guards more complex but instead choose a set of actors such that the actor with the strongest guards, in that sense that a guard being evaluated to true in that actor implies a specific behavior in the other actors, makes the evaluation of the guards of the other actors redundant.

The reason is evident from the combinatorics of the guard conditions. An actor with n guarded actions needs, with an even distribution of input values matching the different guards, on average to evaluate $c_n = \frac{1}{n}\sum_{k=1}^{n}(k)$ guard expressions [1]. If two such actors have either identical guards or the guards check the same property of the input, then the number of checks for the two actors can be reduced from $2c_n$ to c_n. If the guards does not check the same property of the input, the combination of the guards would mean that the number of guards is n^2 where each guard consists of one condition from each actor, this would result in an average of $c_{n\times n} = \frac{2}{n^2}\sum_{k=1}^{n^2}(k)$ which is larger than c_n when $n > 1$.

There are, of course, special cases where guards partially overlap and the number of needed guards would be larger than n and smaller than n^2, but as the trade-off in such cases would be platform dependent, the choice must be left for the designer. An example of this can be found in Chapter 7, where

[1] One could argue that only $n-1$ guard evaluations are needed for n actions as the last guard obviously must be true if the previous was not, in any case, this would not affect the result

one of the case studies is constructed from guards which partially overlap. The problem, however, is that it is not trivial to construct the guards, as is the case when the guards of a single actor is used.

3.3 Nondeterminism and Time-dependent Actors

The other type of dynamic behavior is when an actor is nondeterministic. In many popular programming languages, there is no such thing as nondeterminism, and, nondeterminism does introduce some problems, as two runs of the program may produce different results but still be correct. In an imperative programming language, for example, an alternative statement has a defined order in which expressions are evaluated. The statement

```
if E_1 then S_1
else is E_2 then S_2
```

will only evaluate the guard represented by E_2 if E_1 is evaluated to false. This means that there is an implicit negation of E_1 in E_2, which makes the two alternatives mutually exclusive. In a guarded command based language, such as Promela, however, the statement

```
if
:: E1 -> S1
:: E2 -> S2
fi;
```

is nondeterministic if the guard expressions are not mutually exclusive. In such a case, the language does not define which of the commands should be executed, instead, either choice is correct.

The introduction of nondeterminism in a programming language may introduce some confusion for a programmer, at the same time, it enables the programmer to not specify the program to much. Dijkstra included nondeterminism in *guarded commands*, after stating that he "had to overcome a considerable mental resistance before [he] found [himself] willing to consider nondeterministic programs seriously" [45]. However, he added that he "could never have discovered the calculus before having taken that hurdle: the simplicity and elegance of the above would have been destroyed by requiring the derivation of deterministic programs only".

How, and if the nondeterminism is removed in a final implementation was left to circumstances.

Lee and Parks list five methods in which nondeterminism can be added to Kahn networks. The first two methods, which are available in CAL, are by 1) allowing processes to test input channels for emptiness, and 2) allowing processes to be internally nondeterminate. Methods three and four are to

allow more than one process to write to respectively consume data from a channel. And last, by allowing processes to share variables. [83]

As mentioned, the nondeterminism we deal with in CAL is the first two methods. That is, 1) one action may become enabled if an action with higher priority is waiting for input, or 2) two actions may be enabled simultaneously. We distinguish these two kinds of nondeterminism from each other as the first is only nondeterministic in the sense that it has different behavior depending on when input arrives while the other one is internally nondeterministic.

Definition 3.4. *Time-Dependent Actor. An actor is time-dependent when a lower priority action requires fewer tokens than a higher priority action and their guard expressions are not mutually exclusive; hence, the behavior depends on the time when the tokens are available [124].*

From a scheduling point of view, looking at one actor, we may find a correct schedule for the actor which may not be correct in a certain context. Then again, a nondeterministic actor may in composition with another actor produce a deterministic composed actor. The scheduling problem, hence, is to determine which actors can be composed to deterministic actors such that the nondeterminism and scheduling decision are resolved at design time.

For many-core implementations, it may not always desirable to remove nondeterminism. Instead, this can be used for improving load balancing as an actor can work in one mode as long as input is available but switch at another mode, and do some low priority work, when there is no input available. In any case, the goal here is to remove as much nondeterminism as possible, and where it cannot be removed, e.g. where it is used to handle variable data rates, it will be kept as it is.

3.3.1 Introduction of Time-dependency by Composition

"A composition of time-independent actors cannot be allowed to become time-dependent."

A composition of two actors can incorrectly become time-dependent if the guards chosen for the composition are not strong enough. Consider the example in Figure 3.7, which shows two actors which have input dependent guards. According to the analysis presented so far these two actors can be composed and jointly scheduled, as the actor to the left decides which action will fire in the actor to the right; and this is true. There is, however, one problem, when we consider the two schedules for a potential composition: (s, a) and (t, b), which can be chosen based on the guards of the left actor. The composed actor (see Figure 3.8a) now has two input ports: I and Y, which means that the lower priority action has less input dependencies than

the higher priority action while the guards are not mutually exclusive, and thus, the composed actor becomes time-dependent.

The composed actor will behave incorrectly if there is no inputs available on input port Y. While the guards of the right actor are mutually exclusive, these are redundant, as the outputs of the actions of the left actor exactly match these. The guards of the left actor, on the other hand, do not need to be mutually exclusive, as the token rates are identical for both actions. The composition, however, changes the token rates, resulting in incorrect behavior.

This problem can be solved by making the guards of non time-dependent actors stronger. As this cannot be done to already time-dependent actors, without changing their behavior, the first step is to identify time-dependent actors; an approach for this has already been presented by Wipliez et al. in [124]. Now, when the actor has been shown to not be time-dependent, the guard can be strengthened as follows: each actor adds to its guard the negation of each of the guards of the higher priority actions. In the example in Figure 3.7, this would mean that action t would get a guard $not(i > 0)$, which also would make the composition of the two actors based on the guards of the left actor correct. The resulting composed actor is shown in Figure 3.8b.

What is important in this context is that the guards of an actor that is not time-dependent can be strengthened while the guards of an already time-dependent actor cannot, as the behavior may change. The simple solution is to avoid time-dependent actors in compositions. On the other hand, scheduling of a partition can be allowed to be time-dependent if the relation between guards and FIFO checks remains untouched, that is, a scheduling decision may be time-dependent only if it is identical to the choice in the original actor. This topic, however, is outside the scope of this thesis.

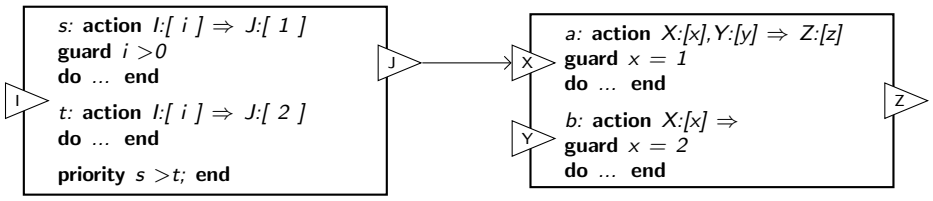

Figure 3.7: Two actors for which composition is attempted. The left hand decides which action will fire in the right hand actor, however, a composition where the guards of the left hand side actor are directly used will result in incorrect behavior as the composed actor becomes time dependent.

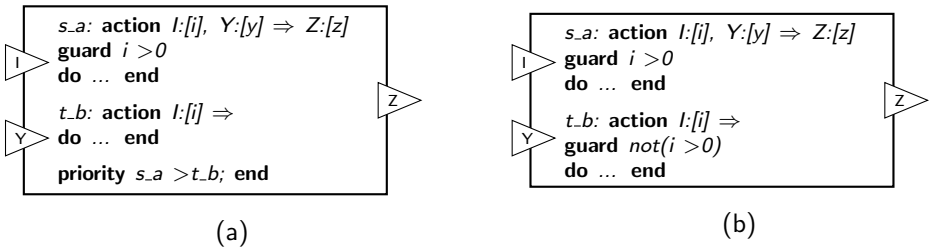

Figure 3.8: The composition (a) using the guards of the *left* actor, introducing time-dependent behavior as the guards are not mutually exclusive and the lower priority action requires fewer input tokens, and (b) with guards that has been made mutually exclusive by transforming the priority to a guard of the second action.

3.3.2 Analyzing Independent Data Paths

"For a partition of actors; if it can be split in to two groups of actors such that the first group has a mode where it can fire an infinite number of actions while the second group fires an arbitrary number of actions, then the first group cannot be scheduled based on the second group"

One more property which is important for keeping the dataflow program consistent regarding token rates is that a jointly scheduled partition should internally have dependent token rates. What this means is that a partition that is to be composed, should not have two sub-partitions which need independent scheduling, or, so to say, have independent data paths. In practice this means that the inputs to the partition must be synchronized by having a token rate dependency. This problem typically appears when separate data streams pass through a partition and the streams are not related to each other.

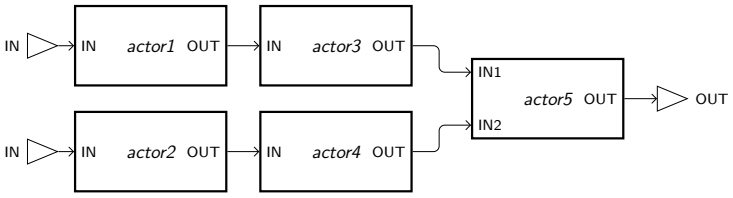

Figure 3.9: Partition with two independent data paths, if *actor5* does not specify a specific rate between its inputs but rather resends whatever is available on either input, this partition is not valid because the input rates cannot be derived from within the partition. If the input rates of a partition are not known it is possible that one of the input buffers would start to grow.

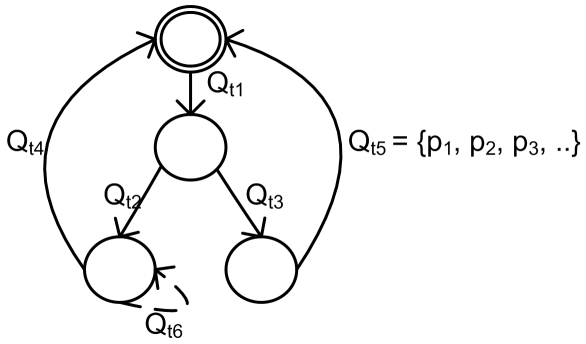

Figure 3.10: Each scenario corresponds to an FSM where each transition has a port access pattern.

The partition in Figure 3.9 shows an example of an actor (*actor5*) that simply resends any input on its output queue; if one input queue is empty it will simply switch to the other one. The two data paths in this example are clearly independent as the data rate on one path is not dependent on the other, i.e. there is no synchronization between the input streams. We can find several static schedules that seem correct for this partition but which may result in deadlock because of either a full or empty input queue.

To find if the data paths are dependent we can again view the partition as a graph where actors and partition input ports are vertices and FIFOs are edges, if an edge is not *synchronizing* it is removed from the graph so that the resulting graph represents the data path dependency. The final test to determine whether the data paths in the partition are dependent is to check that the graph connects the input ports of the partition. As an example, in the graph in Figure 3.9, the edges entering *actor5* would be removed resulting in a graph where the two inputs are not connected. For this to be useful we need to carefully specify which edges are to be removed.

Definition 3.5. *Synchronizing Queue. A queue is synchronizing if it ties the behavior of two actors such that there are only finite firing sequences where the actors are independent.*

Definition 3.6. *Synchronizing Port. A set of ports of an actor are synchronizing if the token rates of these are tied to each other such that, in the long run, the rates are proportional. An actor can have several distinct sets of ports which are synchronized.*

For a FIFO to be synchronizing, it must be possible to derive the token rates of the ports connected to it from the firing rules of the partition. The graph in Figure 3.10 describes an actor as an FSM which has two alternative paths and one state with a self loop. The transitions in the figure are

decorated with identifiers describing the set of ports which are accessed by the action represented by the transition. In this example, the FSM has two states with more than one outgoing transitions. The choice between transitions in these cases can be one of the three following alternatives, 1) depending on input, 2) depending on the internal state of the actor, and 3) a nondeterministic or time-dependent choice. The two first cases are actually identical, as control tokens are required to be described by the firing rules of the partition, and each port of an actor with only such behavior is considered synchronizing; with the assumption that each port of the actor is used is some action that eventually will fire.

The ports that are synchronized by the example actor in Figure 3.10 can then be described as the all the ports of the FSM.

$$Q_a = (Q_{t1} \cup Q_{t2} \cup Q_{t4} \cup Q_{t6}) \cup (Q_{t1} \cup Q_{t3} \cup Q_{t5})$$

When the choices between the alternative paths are nondeterministic or time-dependent, the situation is more complicated. The simple solution is to consider the ports accessed by such an actor to not be synchronizing as the token rate may be arbitrary; each of the FIFOs connected to such an actor are added to a set Q_{nondet} which are always considered to be non-synchronizing. A special case is, if an actor with an FSM has actions which are outside the FSM, if the actions in the FSM are deterministic then only the ports used by the actions outside the FSM are added to Q_{nondet}. The set of queues that are two way synchronizing between actor a and b can then be described as:

$$Q_z = (Q_a \cap Q_b) \setminus Q_{nondet}$$

This is a more conservative approach than what is allowed by Definition 3.6 and in many cases it is possible to find one or more subsets of the ports of an actor that are synchronizing. An example of this is two ports that are accessed by the exact same set of actions where for each action have identical token rates on these ports, and therefore, obviously have related token rates.

What is described by Definition 3.6 is actually that the internals of the actors can be represented as a a graph, where a time-dependent actor may produce a disconnected graph. The graph is constructed by connecting ports which has proportional token rates:

$$\begin{aligned} Q_a^{full} = &\{(p_1, p_2) \in P_a \times P_a | n_{p_1} \propto n_{p_2}\} \\ &where \\ &n_i = \text{ number of tokens on port } i \end{aligned} \tag{3.2}$$

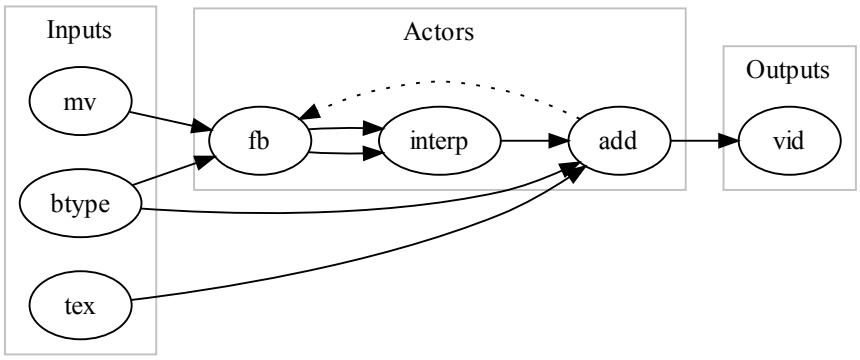

Figure 3.11: Graph showing the motion compensation network of an MPEG-4 decoder after non-synchronizing edges have been removed and are shown as dotted arrows.

For an actor a, the set of synchronizing ports Q_a are described by one of the connected subgraph of Q_a^{full}. This property is rather generic and can be proved in the general case for the actors which are time-independent, however, for actors with time-dependent behavior, the property can be proved for certain cases. For this reason it is not further explored in this thesis, instead the rest of this thesis assumes that time-dependent actors are identified, with methods such as [124], and the queues associated with these actors are considered to be non-synchronizing.

Figure 3.11 shows a graph representing the data paths of a sub-network. The edges that have been identified to be non-synchronizing are represented as dotted arrows. The direction of the arrows is of no importance and simply show the direction of the dataflow. As can be seen, there is a path from each input port to the other input ports, which means that the inputs are related to each other and the token rates are synchronized. This means that the partition does not have independent data paths, and can be scheduled as one unit.

3.4 Correctness of Scheduling Models

A scheduling model can be considered correct when it describes the inputs of the model completely as a set of finite data types and has a set of guards, based on the guards of the original actors, corresponding to each of these inputs. The individual input streams of the actors of a partition should therefore be seen as one single input stream, simply using the separate channels to feed data to different calculations. The guards, identifying the input sequences, based on which a schedule is chosen, may check properties of the partition as well as input values. As the composition of actors is supposed to simplify scheduling decisions, one actor is chosen to lead the partition while

the other actors are forced to follow. This results in that some properties must hold, regarding the *leader* actor and the rest of the partition, for the scheduling to be successful.

This results in the following type of compositions: 1) Checking input values is only allowed for one actor per partition – the partition leader. Other actors may only use control tokens indirectly by reacting to action firings in the leader action. 2) The leader actor must enforce static token rates on the other actors' external inputs. It cannot allow another actor to run an arbitrary number of times without causing buffers to overflow. 3) The partition may work with more data at the time compared to the original program – a schedule instead of an action firing. However, the partitioning is not allowed to introduce time-dependency.

3.4.1 Boundedness of Nondeterminism

There can be two variants of nondeterminism in an actor partition. The first one, which was already discussed earlier in this chapter, adds flexibility to the order in which actions fire or to the token rates of the actors. For this, it is enough to show that the partition does not have less flexibility on its inputs that the original program. The other type of nondeterminism, the one of interest in this section, is not related to what sequences a partition accepts or if the partition is deadlock free or not, instead it is more related to the semantics of the programming model.

As an example, consider an actor which has a low priority action a_l which is used to optimize some parameter of the actor. This action have neither inputs nor outputs but simply performs some calculations to tune the actor, but does not change the state of anything in the actor related to scheduling. Now, when the actor performs one iteration, it fires a sequence of actions e.g. a_1, a_2, a_3, consumes an input sequence S_{IN} and produces an output sequence S_{OUT}. Such an iteration may include the action a_l but it is also correct behavior if it is not fired. Now, if the firing sequence can be fired repeatedly for any possible input, one simple SDF schedule can be used to schedule this actor, however, and this is the question: is it correct behavior when such a schedule implies that a_l will either fire once per iteration (if it is included in the schedule) or never (if it is not included in the schedule)?

Technically speaking, the scheduling model does not guarantee unbound nondeterminism. The reason should be obvious; the purpose with scheduling is to choose a minimal number of firing sequences from the great number of possible alternatives for interleaving actions that are possible. This in turn, means that a more regular firing pattern is enforced which in turn means that nondeterministic behavior becomes deterministic.

3.4.2 Valid Partitions

Finally, for a partitioning to be valid, the actors in the partition must be allowed to fire without requiring actors outside the partition to be interleaved. When a scheduling model is constructed from an actor partition, it is assumed that the required inputs to the partition will become available simply by waiting for other actors outside the partition to fire enough times. The problem arises when the actor that is supposed to produce an input waits for tokens from an actor inside the partition. As the scheduling model is required to completely describe the behavior of the partition, we cannot allow this type of behavior as the required interaction between the partition and the surrounding actors affects how the partition must be scheduled.

We need something similar to the composition theorem by Pino et al. [105], which states that an SDF composition is valid if it has an acyclic precedence graph. While this perhaps is unnecessarily strict for a CAL composition, we can use a variant of it to guarantee correct partitioning. Consider a simple CAL program with the four actors a, b, c and d. Actor a has a channel to both b and c, while actor d receives tokens from b and c. Based on the topology, a preorder can be described as $\{(a \leq b), (a \leq c), (b \leq d), (c \leq d)\}$. If a partition, p_{abd}, including the actions a, b, d is proposed, the precedence will become $\{(p_{abd} \leq c), (c \leq p_{abd})\}$, indicating that action c both precedes and succeeds the partition. If the partition is scheduled with the connections to the actor c as simply inputs and outputs to the partition, there is no guarantee that the schedule will be correct.

In order to construct correct partitions, partitions must be constructed such that actors outside the partition either precede or succeed the partition, or are independent. This approach is conservative as it omits the actual behavior of the actions and simply uses the actors and connections to decide if a specific partitioning can be allowed. There will certainly be cases where this rule is to strict and will forbid partitions that would work correctly. The goal here is instead to give a sufficient (but not necessary) condition for that a correct scheduling model can be constructed for a partition.

Chapter 4

Analysis of an Actor in Isolation

Even though the target is actor composition, much useful information can be extracted by analyzing single actors in isolation. The scheduler of a single actor can often be simplified which as such can improve the generated code, but, of course, the goal is to simplify the actors before attempting to compose the actors. The actor schedulers typically have similar constructs as the control flow of an imperative programming language, however, more explicitly described. Consequently, transformations resembling what is usual in most compilers, such as loop unrolling [7], can be performed on the actor scheduler as well. Actor-level scheduling is useful for the scheduling models as it groups single input tokens into groups that are to be processed together. A schedule thereby gives the token rates of the actor and is represented as a pair consisting of a guard expression and a sequence of action firings. For a network partition, the schedules of the front-edge actors drive the scheduling by defining the portions of data that can be processed together, then, typically, it is possible to calculate balance equations between the schedules in order to derive the inputs of the partition.

The actor scheduler can always be seen as an FSM, with guards on some of the transitions, and where the transitions (in form of actions) modify the state of the actor [1]. The first thought on analyzing the action scheduler is to identify transitions that are input dependent and to identify loops. On second thoughts, while simple loops are easy to identify from an FSM, for more complex schedulers it is more difficult do define what a loop actually is. Consider as an example the FSMs in Figure 4.1; in the first example (a) there are two clear loops, the self loop in the state σ_3 and the loop covering the whole FSM, in the second example (b) the loops are not that

[1] Actors without an explicit action scheduler are simply seen as an FSM with a single state

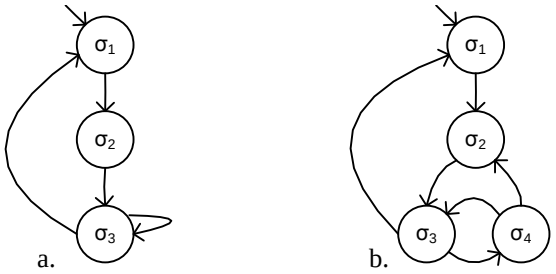

Figure 4.1: Two examples of action scheduler FSMs.

clear. To handle every action scheduler in a general fashion, a more general approach is to simply investigate if there is a static path through a part of a scheduler, and not really care about what the FSM looks like. In practice, this means that if some states that can be seen as starting and ending points of schedules can be identified, the paths between these should be found and be verified to be unique.

In this chapter the focus is on what can be found from analyzing an individual actor. The goal is to produce sufficient information for handling the state space of actor compositions and to reduce the number of states needed to describe the scheduler of the composed actors. By more accurately specifying the actor state from a scheduling point of view, and by potentially reducing the number of states the action scheduler (the FSM described in CAL code), the analysis can be focused on the few choices that link the behavior of the actors that are to be composed. Finding out which scheduling decisions can be resolved at compile-time can be seen as a first step to scheduling a composition while the next step then is to find if further decisions can be removed using what is known about the interaction of actors, and finally the actual scheduling is about interleaving the actions of the set of actors in the composition.

Goals with Actor Level Analysis There are two goals with the actor level analysis. One is to describe the state of an actor. While this, to some extent, has been discussed in Chapter 3, we can do better. The actor state should be described with the smallest possible number of variables to make the scheduler efficient, but enough variables to describe the scheduler correctly. This means that some variables are part of the state in some FSM states while are irrelevant in other FSM states. The second goal is to describe the actors as a set of action firing sequences that can be proved to correctly describe the behavior of the original actor; these actor level schedules then serve as one of the input to the partition level analysis presented in the following chapter. It is important to identify these as the schedules of the composed actor should start and end in states where a run-time decision

is needed; the actor level schedules are for this reason used to predict the scheduler of the partition. As these two goals are strongly related, the following sections will jointly discuss both topics and the approaches that can be used to generate this information.

4.1 A Simplified Actor Scheduler

The presented approach can loosely be described as three steps that can be repeated until a valid model is found; the following steps can be identified: 1) Create a high level description of which schedules optimistically are expected to exist, 2) Generate the concrete action sequences representing these schedules, 3 validate that the schedules are unique or if necessary split the schedules that are not unique and redo the steps. The approach used resembles abstract interpretation [99, 124], but technically, the model is restructured before the interpretation to not include any code that would use variables with unknown values, but only values that may not be unique.

The starting point is an actor, which is viewed as an FSM where transitions withe guards that depend on a value from outside the actor are marked as an *input dependent transition*. A guard is considered to be input dependent if a input port (or a value that is returned from a native function) is reachable from the guard in the variable dependency graph. These transitions are considered as dynamic scheduling decisions, and are, as in the example in Figure 4.2, marked with a question mark, indicating a point where a synchronization with another actor is needed. Before starting the actor level scheduling stage, the actor is simplified to only include the instructions and variables that are related to scheduling. This is done simply by removing variables and instructions that are not reachable from any guard in the actor partition. As a result, the actor will look like the FSM in Figure 4.2 (left), only including a few variables, in this case the variable named x.

The idea needs some clarification before continuing with more details. Consider what using abstract interpretation means for analyzing scheduling of a CAL actor; in practice it means that input tokens produces unknown values in the actor, which if used in calculations are fine but if used in guards means that it is not possible to know which action to fire next. What is the goal here is, that, when a state where the guards use such an unknown value is found, each of the outgoing transitions are examined. It could be thought of as, if this guard would evaluate to true, then this would happen. When a guard is, so to say, forced to evaluate to true, even if the values used in the guard are unknown, every variable used by that guard must be set to unknown in order to not enable some impossible behavior further on in the interpretation. How this is actually done in the implementation of this

work, is that a set of schedules are identified that are expected to occur then abstract interpretation is used to find the exact sequences corresponding to these schedule, and finally the schedules are verified to be unique.

Compared to abstract interpolation, in the approach used here, the guards where unknown values are used are already identified beforehand which means that evaluating an guard with an unknown value will never occur. The approach also have another difference, when a state with an input dependent choice is found, the approach tries to run each of the possible branches ignoring the guard with the unknown value. The approach can be seen as a less strict version of abstract interpretation, which when possible fires actions according to the guards but when this is not possible tries to fire each possible action of that state. How this in done is practice will be described in the following sections.

4.2 A First Approximation

Figure 4.2 shows the transformation that is desired for each of the actors. The original actor has an initial state named *idle* in which one of the actions a,b, or c can fire depending on some input value. In the other state, named *work*, there is a loop which will fire 63 times before the FSM returns to the *idle* state. Now, the goal is to transform this actor into the second FSM shown in the same figure. In this simplified action scheduler, there is only one state and four transitions; while the input dependent guards are still present, the guards related to the loop has been removed. The transitions are no longer corresponding to an action but instead to a sequence of actions that can fire as a result of a guard being evaluated to true. The new scheduler will for each guard evaluated to true, run a sequence of 65 actions without

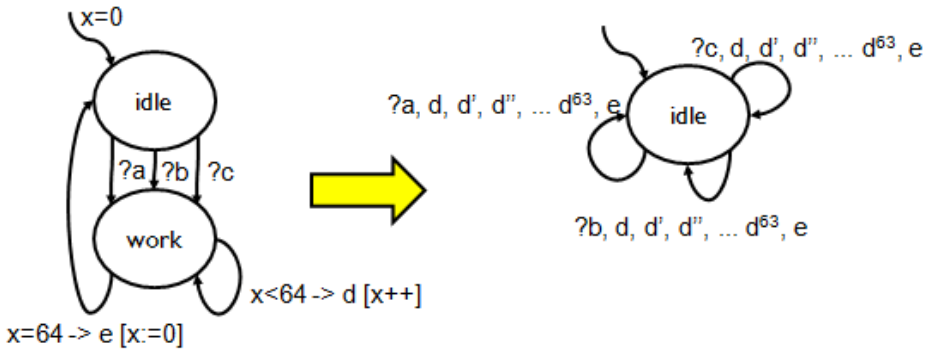

Figure 4.2: An actor can often be described in a simplified form where some actions sequences are resolved compile-time.

evaluating any guards in between.

An approximation of the simplified scheduler can be derived by running a reachability test on the action scheduler while systematically checking the guard dependencies of the transitions taken. We define the *choice states* as the set of FSM states in an action scheduler, where one or more of the guards depend on an input port (directly or indirectly); also the initial state is included in this set no-matter if it actually is a real choice state or not. These two types of states are suitable for starting and ending schedules for two different reasons. In the first case, returning to the initial state means that we are in a state where continuing would only be repeating previous schedules. In the other case, requiring an input value, the actor is in a state where it needs a value that is not available in the actor at compile-time. The interesting schedules to be searched for is the set of paths in the FSM connecting the *choice states*, consequently, for each such state, a separate schedule is needed for each outgoing path.

The abstract schedules are constructed by performing a depth first search on the actor FSM, constructing a tree, terminating at either an allowed end state (a choice state) or an already visited state (representing a loop in the schedule). In the example FSM in Figure 4.3, three schedules, corresponding to the three input dependent transitions, leaving the choice state *idle* and finally returning to the same state, are expected to be found. The information at this point regarding the schedules is the two states between which the schedule defines the path and the action that the schedule starts with. For more complex actors, the depth first search may find several reachable choice states which are potential end states of the schedule. In practice, for a schedule to be static, it can only have one single end state, while the other potential end states are seen as unreachable at the point when the actual schedule is found.

When the FSM has choice states that are not the initial state, the exact

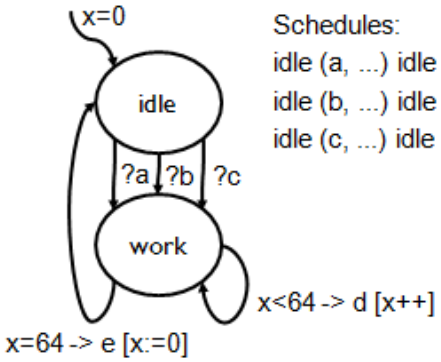

Figure 4.3: The schedules to be found.

state where the schedule starts needs to be known in order to analyze the schedule. By sorting the schedules such that a schedule starting at a non-initial state is not analyzed before another schedule that ends in that state (named *mid* in Figure 4.4), a valid state is guaranteed to be known. This does not mean that that state is unique, instead the validity of the state and the schedule must be verified in a later step. At this point, what we have is an abstract representation of the schedules that are expected to describe the behavior of the actor.

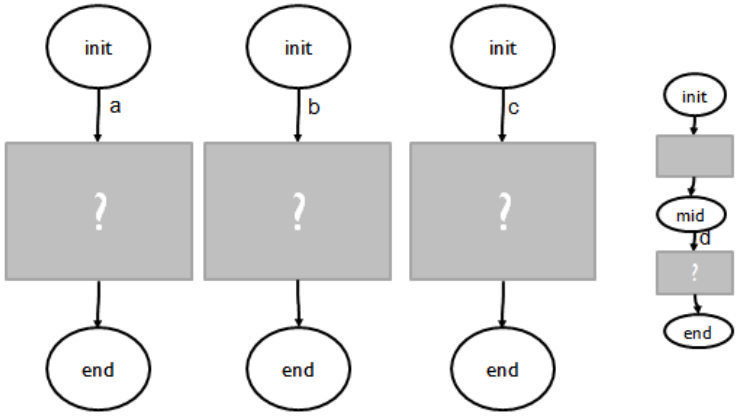

Figure 4.4: Input to the Abstract Interpretation.

4.3 Concrete Schedules

The concrete schedules can be found by simulating the actor starting from each of the choice states with the schedules corresponding to outgoing transitions of that state. For this, the implementation of abstract interpretation (available in Orcc) is used, and is extended to allow forcing the execution to start with a specific action without respecting the guards. This feature is used to make the interpreter run a specific schedule in those cases when the schedule is known to start with a action which guard depends on an input value. This may feel weird as an action may be fired even when the guard would evaluate to false, especially considering that the some of the ignored values may be reused later in a guard expression inside the schedule. However, it is not a problem as, if the input value is reused in a guard, that state will be considered a separate choice state, and, if the ignored guard except from checking an input value also checks some other state variable which also would be used in a guard within the scheduler, the schedule would be found to not be unique as this variable seemingly does not have an unique value for this schedule which means that the schedule would be split into

Algorithm 4.1 Scheduler pseudo code.

```
1: schedules := buildInitialScheduler()
2: for all s ∈ schedules do
3:     interpreter.setNextPath(s.enablingAction)
4:     interpreter.initialize(s.initState)
5:     repeat
6:         interpreter.schedule()
7:         s.sequence.add(interpreter.lastAction)
8:     until (interpreter.state ∈ choiceStates)
9: end for
```

two schedules.

By the definition above a schedule is a sequence of action firings between two states where the scheduling decision cannot be performed at compile-time. From this follows that the schedule does not have any guards depending on *unknown* values, except for possibly the first action of the schedule which represent the scheduling decision enabling that particular schedule. Abstract interpretation allows the program to work with unknown values which enables to view the data streaming through the actor as valueless tokens, while only variables needed for guards must have concrete values. Unknown values can also be allowed in guards of actions corresponding to outgoing transitions of a choice state, as these guards are defined as dynamic (run-time) decisions. What this means for a practical implementation is that the abstract interpreter is instrumented to run a specific schedule, and in case the schedule starts with a dynamic choice, the interpreter is instrumented to, without evaluating any guards, start by firing that specific action that has the guard that enables the schedule.

The operation of the scheduler can be described as in Algorithm 4.1. Basically, the interpreter schedules and runs one action at the time until a choice state is reached. The interpreter is first initialized with the state that corresponds to the starting point of the specific schedule and in case the schedule starts at a state requiring a choice of more than one transitions, the interpreter is instructed to start the execution by forcing a specific action guard to evaluate to true.

What has been achieved by this step is that the actor is described by a set of static schedules that are valid execution paths between the choice states for the specific state the actor was in when the schedule was generated. If the schedule passes through states with more than one outgoing transitions, the state is considered to be a potential choice state until it has been proved that the schedule is an unique path between its initial and end state. If it cannot be proved that the choice between transitions in this state is unique, the state must be seen as a choice state and the schedule is not valid. Instead,

the analysis must be repeated with a new initial scheduler including the new choice state. These steps are then repeated until every schedule is proved to be static, which in the worst case means that we end up with the original actor with one action per schedule.

4.4 Validation of Action Sequences

A schedule is always described with respect to a context (closure or signature) which for a CAL actor is a subset of the state variables, the FSM state, and the contents of the input queues. Naturally, this set only needs to contain variables and input ports which are reachable from a guard expression of one of the actions in the schedule. The schedule is fired based on evaluating the guard of the first action of the sequence, while the other guards are expected to always evaluate to true once the schedule has been chosen for execution. Now, the variables of the guard expression of the action enabling the schedule, may not define the context uniquely enough for the sequence to always follow that specific guard. The guard may not restrict one variable enough or it may even ignore that variable; the sets of variables related to scheduling can be seen as: $Guard \subseteq Context \subseteq ActorState$.

The schedules generated according to the approach described above, was scheduled with a correct but arbitrary context with respect to the state variables. The generated schedule can certainly describe a correct action sequence for that particular state, but, the guard may not be strong enough to decide if the context exactly corresponds to that schedule. The approach taken here is not to strengthen the guard, but instead to shorten the schedules such that the context corresponds to the guards. This is done by inserting new choice states when a state is found where the guards of the outgoing transitions are too weakly described in the guard enabling the schedule. To find these states, the schedules are simulated with respect to how the actions access and modify the state variables.

For each action, what we are interested in is, mainly, the instructions storing a value to a state variable. A store instruction, in the IR (Intermediate Representation) we are working with, stores a value based on an expression, composed of constant values and local variables, to a state variable. By using the variable dependency graph, based on the instructions of the specific action only, a store instruction can be described as a modification of a state variable, from a set of state variables (and input ports). When repeating this for every store instruction, a schedule becomes a sequence of action firings which modify the state variables from other state variables. Some of the modifications are completely based on constants, which mean that the state variable is reset with a constant value. Other variables are modified based on some other variables or in some cases themselves. When

a variable is updated from variables, the variable inherits the *state* of the variable, which at a specific moment either is known or unknown. Figure 4.5, shows an example of how a store instruction is described from the program description.

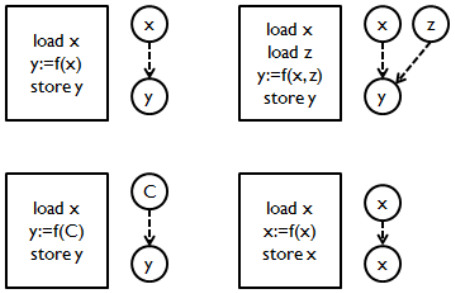

Figure 4.5: Read/write analysis.

With this information, it is possible to, for each action, 1) give a set of variables that are reset, that is, gets updated with a constant value, 2) give a set of variables that are updated, including from which variables they are updated. By stepping through the sequence of actions, it is therefore possible to keep track of which variables has a known value and which does not. In the simplest type of analysis, we start with considering the variables in the context of the schedule to be unknown (or dirty). Then, for each action, if a variable is updated from known (or clean) values, where a constant always is known, then the variable itself becomes known. At the point where a variable is used in a guard, it can only be allowed to be based on known variables, else the state with that guard must be considered a choice state. In order to verify that a scheduler is complete, each of the schedules is run with this check to find out if there is a guard in one of the schedules which uses a variable with a potentially unknown value. If this check holds, we can be sure that the schedules are unique and are eligible when the action with the enabling guard is eligible; but, of course, only if sufficient input for the whole schedule is available.

A variable does not always need to start out as unknown. If every path to the state where a schedule starts resets the variable to a specific value, such that the variable always has the same value in the initial state of a schedule, the variable can be seen as known from the start, when that schedule is analyzed. To find these variables, two properties are checked for each variable belonging to the context of a schedule. First, a variable can only be known if it can be proved to always have a constant value in the state where the schedule begins. To prove this property, each path to that state must reset the variable such that it is *clean* when the state of interest

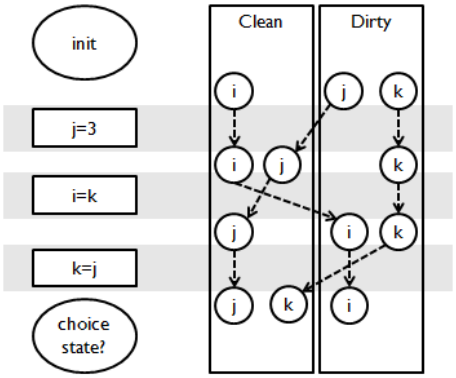

Figure 4.6: Validation of uniqueness of schedules.

is reached; this can be resolved with the same analysis as presented above. The second property is that each path to that state should end with the variable having the same value, including the initialization of the variable if it corresponds to the initial state. To achieve this, the interpreter producing the schedules records a snapshot of the state variables after each schedule it has executed. The set of variables that are both clean when reaching the scheduling state and have an identical value, no matter which path was taken to the state, is considered to be clean from the start of the analysis. Also, if the guard of the schedule is strict, the variable might be more or less known during the schedule. This, however, might quickly become complex when the variables are modified, and have not been utilized in this work.

In many cases, for example with a variable used to iterate a loop, a schedule either resets the variable before it starts looping or afterwards. In both cases, the variable is reset before it reaches a guard; in the case when it is reset before it is used, the analysis finds the variable to be known when the schedule is checked, in the other case, the variable should be marked as clean in the context of the schedule, if it can be shown to always have the same value in that FSM state. A simple example of how the bookkeeping of state variables works is given in Figure 4.6, which shows a schedule with three actions, each containing one instruction modifying the a state variable. The first instruction updates variable j from an constant which makes j known. At the potential choice state, variables j and k are known while i is unknown. If the guards of the potential choice state does not use i, the choice of the next action to fire is a static scheduling decision, else it is not.

This analysis can be improved by describing more carefully the relationships between guards and variables, and states and variables, however, it must be conservative in that sense that a schedule is only considered static when it can be proved to be. Consequently, a static schedule may be split

into several schedules, because it could not be proven to be unique. The resulting scheduler, after this step, describes the behavior of an actor with a simplified scheduler where the scheduling decisions that can be resolved at compile-time have been removed and replaced with sequences action firings. If the action sequences are composed in to bigger actions (super actions), the new scheduler causes the actor to work on larger chunks of data, which may cause problems especially if the program includes feedback loops. This is similar to the known problem that deadlock-freedom, for models such as SDF, is not closed under composition. Instead, composing two adjacent actors may introduce deadlock, and deadlock-freedom needs to be proved according to composition conditions such as those presented in [105].

The conclusion from this is that while the schedules are correct, for a composition to be deadlock-free, it may be unavoidable to interleave actions of two schedules from two different actions. This is exactly what the composition presented in the next chapter is about, and the composition presented there does guarantee that a composition is deadlock-free, however, it does not guarantee the program to be deadlock-free globally.

4.5 Describing the Actor State More Precisely

When an actor is described as a set of static schedules, the accesses and modifications of state variables become easier to analyze. For the different choice states, a subset of the state variables represent the context of the schedule. When the value of a variable cannot reach any of the guards in a schedule, typically because the variable is reset before being used or because it is not used at all in the guards of the schedule, these variables should not be used for describing the state of the actor in that specific FSM state. In other words, when the actor is in a specific FSM state, the actual state of the actor is described by the FSM state and the state variables, however, only the variables that affect the choice of the next schedule, in that FSM state, should be used in order to avoid introducing unnecessary states.

What should be resolved is which variables are relevant in each particular FSM state that corresponds to a choice state. The benefit from describing a state more precisely will become more evident in the next chapter, when a scheduler for a partition is generated. Then it is essential that any state is not duplicated in the scheduler as every *new* state, results in a more complex scheduler. When there are two states, where the future action firings will be identical, the two states should be considered the same state. To find the variables that are relevant for a state, and to remove those which are not, an analysis similar to the check of uniqueness of schedules is performed. The difference is that it is required to verify the value reachability beyond the schedules only. Furthermore, with composition, it is not only required

```
<actor name="dequant">
        <fsm initial="start">
        <transition action="get_qp" dst="start" src="start"/>
        </fsm>

        <superaction name="start_get_qp">
        <guard/>
        <iterand action="get_qp" repetitions="1"/>
        <iterand action="ac" repetitions="63"/>
        <iterand action="done" repetitions="1"/>
        </superaction>
</actor>
```

Figure 4.7: The actor level scheduler representation which is generated for each individual actor.

to check the values that ends up in a guard of the actor, but also, values that are sent to another actor which in turn uses it in one of its guards.

The question that this analysis answers is: In this state, which variables will be reset before they are used in a guard expression or are used to produce control output. Then, when comparing two actor states, the variables which value will not be used before the variable is reset, will not be used to compare the states.

4.6 The Results of the Analysis

The analysis presented in this chapter is used to schedule the individual actors, such that an actor is described as a set of static schedules, which are fired based on the guard of the first action in the schedule. This is useful for predicting the schedules to be constructed for the actor partitions, described in the following chapter. The reason for why this in needed, is that, in order to search for a schedule, we need to define how much work the schedule is supposed to perform; this can be in terms of how much input tokens are consumed or a partial description of which actions must fire during the schedule. By first scheduling the individual actors, the static firing sequences of the actors (see Figure 4.7) gives a limit of how many actions can fire before a dynamic guard expression must be evaluated, similarly the token rates of these sequences and the more precise descriptions of the states are known as a result of this.

In practice, this analysis is performed as a set of analysis passes run by the Orcc compiler as part the Promela code gereation. In addition to generating the Promela code, also some additional information, which is used by the scheduling tools outside Orcc, is generated. While, how this

```
<actor name="dcpred">
        <schedule initstate="read" action="start">
                <rates>
                <peek port="BTYPE" value="2048"/>
                <read port="BTYPE" value="3"/>
                <write port="START" value="1"/>
                </rates>
        </schedule>
        <schedule initstate="read" action="read_other">
                <rates>
                <peek port="BTYPE" value="0"/>
                <read port="BTYPE" value="1"/>
                <read port="QP" value="1"/>
                <write port="START" value="1"/>
                </rates>
        </schedule>
...
</actor>
```

Figure 4.8: The generated information regarding the token rates and values of the actor level schedules.

information is used is presented in the next chapter, this section presents the actual information generated by the analysis.

The first part, the actor level schedulers, describe the schedules of each of the actors, and more importantly, the states where a dynamic scheduling decision is needed. An example of the generated actor level representation is shown in Figure 4.7; this actor is simplified to an FSM with only one state and one transition corresponding to an action firing sequence. From this actor, it can be seen that one iteration corresponds to 65 action firings, and ideally, a schedule for a partition including this actor also includes the corresponding sequence, possibly interleaved with sequences from other actors.

To construct the scheduling model of a partition, it is also useful to have some additional information regarding the actor level schedules. Figure 4.8 shows the information generated regarding token rates and values that is used to generate the appropriate input sequences for a partition of actions that are analyzed for composition. While the values of *read* and *write* describe the number of tokens consumed and produced on each of the ports when a schedule is fired, the value of *peek* describes the actual value that is accepted by an input dependent firing rule, based on the approach in [124]. Finally, the last piece of information, is the refined description of the actor state which is used to compare if two states of an actor are identical from a scheduling point of view. An example of the this, as it is generated by

```
<actor name="zigzag">
...
    <state>
        <allstates>
            <variable name='count'/>
        </allstates>
        <fsmstate name='full'>
            <variable name='count'/>
        </fsmstate>
        <fsmstate name='empty'>
            <variable name='count'/>
        </fsmstate>
    </state>
</actor>
```

Figure 4.9: Defining the variables that are part of the state of the actor, first *allstates* describes all the variables that are part of the state, then *fsmstate* defines for certain states which variables can be ignored in that specific state.

the compiler is shown in Figure 4.9, where additionally to the FSM, the actor has a variable that describes its state. The first piece of information (allstates) simply describes the variables that are part of the actor state, while the second part (fsmstate) lists the variables that can be ignored in that specific state.

In addition to the Promela code, these pieces of information serve as the inputs to the following phase of the scheduling, where the goal is to generate schedules for actor partitions. This information is then used to predict the schedules that can be used to schedule the partitions and to partially describe the initial and end states of the partitions.

4.7 Related Work

Even though the actor level scheduling only is a step towards actor composition, there are various similar methods that are relevant as a comparison and to some extent has inspired some of the ideas used in the approach presented in this chapter. The methods present quite different ways to represent the operation of a CAL actor, which then either is used for simplifying the scheduling of the individual actor or is used to enable actor composition.

Actor Classification One related work that the approach presented in this chapter, not only borrows ideas from, but also concretely builds on the implementation of, is the approach presented by Wipliez et al. in [124]. This method is based on a classification of dataflow actors into more restricted

but easier to schedule MoCs, such as SDF, CSDF, and PSDF. The approach in [124] is based on abstract interpretation and satisfiability, where a more restricted MoC is decided for one actor at the time and the internal scheduling of the actor is made minimal. The classified actors can then also, on the level of the program, be more efficiently scheduled according to the scheduling principles available for the different MoCs. This means that for a part of a program where every actor can be classified to a statically schedulable MoC, the run-time overhead will be minimal as all scheduling decisions can be done at compile-time [81].

The classification approach uses abstract interpretation to find which scheduling conditions can be resolved from the actor itself and satisfiability to find the different configurations for an actor which fires differently depending on an input value. The abstract interpretation can be described informally like this: the actor is initialized and starts to fire actions based on evaluating the guards of the actions. Variables may have concrete values or unknown values, for example, as a result of a read from an input channel. For the calculations in an action, if the variables used in the calculation are known, then the result from the value is a concrete value, else it is an unknown value. Unknown values are not a problem as long as no guard uses a variable with such a value, which would mean that it cannot be known if the guard will evaluate to true or false.

The approach in [124] uses abstract interpretation as follows. If the actor can be run, from its initial state, back to that state, without evaluating any unknown guard expressions, the actor can be classified as SDF. The actor state here means, the state of the FSM and the state variables that are used in guards. The intuition here is that, for one iteration of the actor, no unknown values reached any guards, and as the initial state is found, the next iteration will be identical, which means that the actor repeatedly will fire this same sequence of actions. When it takes a few iterations of the FSM to reach the initial state, the actor is instead classified as CSDF.

Actors that are input dependent can still have static behavior if depending on the control input, one static schedule can be chosen. Such actors are classified as quasi-static, which in this approach resembles PSDF. The quasi-static actors are considered to have a set of configurations, which maps a control input to a sequence of actions, which themselves can be classified as SDF or CSDF. The control tokens corresponding to the configurations and that are feed to the abstract interpreter needs to be found such that abstract interpretation can be performed for the different configurations. In [124], an approach using Satisfiability Modulo Theories (SMT), which is used to find an unique value that is accepted by a specific guard, but not by any higher priority guard. This method for finding the control values, including the implementation of it, is also used in the work presented in this thesis to find control tokens for guards expressions that references an input queue (a

peek). However, this topic is more relevant for the next chapter, so we can let i rest for a while.

Another property which is handled in the classification approach is time-dependent behavior. CAL allows actors to be time-dependent in the sense that a different action may fire if more input tokens are available, which means that triggering the actor at a later moment may lead to a different behavior. In practice this means that an action requires fewer tokens from the input ports than a higher priority action while the guards of both actions are evaluated to true. In the abstract interpretation, the inputs are considered to always have enough inputs which means that time-dependent behavior is not allowed as the resulting scheduler will loose this behavior. In the variant presented in this chapter, time-dependent behavior should not be a problem as long as the state with this behavior is identified and the choice between the actions is considered a run-time decision. To identify the time-dependent choices, the approach from [124] is directly used as it is available in the Orcc compiler.

Dynamic Code Analysis The actor level schedules are to some extent inspired by what is called actor chains in the work by Boutellier et al. [27]. In this work, dynamic code analysis, which means that the program code is instrumented with operations gathering information about the program when to program is executed, is used to extract static schedules. Static schedules are extracted both on the level of individual actors and on partitions consisting of several actors. In the context of this chapter, we are only interesting in the scheduling on actor level, but we will get back to more details regarding this work in the next chapter.

An actor chain is defined as a sequence of actions that always follow each other. In [27], action chains are obtained by executing actors in isolation according to FIFO traces, which has been recorded from executing the program with training data. The execution traces then, ideally, if the training data is sufficient, covers every possible action sequence that can occur. The detection of action chains is based on data dependent actions, FSM states, or state variable values.

Compared to the analysis presented in this chapter, the detection of actor chains in [27], have the same goal, and constructs action sequences based on very similar conditions. The most significant difference is that dynamic code analysis is based on real execution traces while the approach presented here is based on abstract interpretation. Verification of uniqueness of schedules is, for this reason also different, as in dynamic code analysis, it is based on the nonexistence or alternative schedules, while in the presented approach it is verified based on an approach that can be compared with an analysis of the reachability of variable definitions [99].

Both methods also use the actor level scheduling as an initial step that

enables the scheduling of actor partitions. It seems like both methods for actor level scheduling, ideally, should give identical results. However, the method presented by Boutellier et al., may be less conservative and may find chains that are not detected by the presented method; on the other hand, the method based on dynamic code analysis depends on sufficient training data and would obviously give overoptimistic results if the training data is not sufficient. This said, it seems both methods have their benefits, and the presented approach could still borrow some more ideas from this approach in order to more efficiently detect action chains.

Actor Machines Another way to represent the scheduling decisions of an actor is presented by Janneck in [73]. This model, called *actor machines*, is an automation that implements the behavior of the actor and encapsulates the action selection mechanism in an *action machine controller*, which consists of controller states and *test-*, *execute-*, and *wait-* instructions. Each of the controller states keeps the information regarding which conditions are satisfied, which are not, and which are unknown in that state while the instructions make the automaton proceed to the next state by executing actions, checking conditions, or waiting for input.

The controller states are mapped to a set of conditions that, in that specific state belong to the set $\{true, false, unknown\}$, which means that, the controller states state which conditions are known in that state. The *test* instruction then, which based on some conditions, proceeds to one of two states, is only needed if the condition tests a condition that is unknown in the state preceding the *test* instruction. Furthermore, as the *test* instruction tests one of the conditions, that condition is known in the state reached after the *test* instruction. This means that the same condition does not need to be retested again after it once has been determined [33]. On the other hand, some conditions cannot only be tested once; consider for example a condition that tests if input is available. For this purpose the *wait* instruction is used to erase the information about a *volatile* condition [73], which if at one moment evaluated to false may at the next instant be evaluated to true, as input data may become available.

An actor machine seems to be a practical description of the scheduling process of an actor, and seems to encapsulate the same information as the actor scheduler description that is described in this chapter. The actor machine model, however, includes slightly different information, as it describes more specifically the conditions that are known in different states. From this point of view, the actor machine model might be a good alternative to describing the simplified actor schedulers produced in this chapter to possibly make the action selection process more efficient. The difficulty in both models, however, is to create the model from the original CAL actors.

In [33], Cedersjö and Janneck present a number of translators from CAL

to actor machines, with different properties. The more advanced generators presented allows that actor machine to *memorize* tests such that tests where the result cannot change are not repeatedly performed. This is then used to generate more efficient schedulers for the actors. The actor machines can further be reduced by removing choices where more than one action is available in a controller state. As an example, a controller state may be succeeded by two test instructions that both are mutually exclusive, now, in this case it is unnecessary to keep both test instructions; instead the automation can be reduced.

The benefit from using actor machines in code generation is that the process of choosing the next action to fire can be simplified as the sequence of conditions to be tested is not constructed by chance but is based on knowledge about the actors. By this, it is always known what conditions make sense to check and conditions that are known are not rechecked. This result is to some extent similar to what we achieve with the analysis in this chapter, that is, which conditions are known at which state. This said, the actor machine model could be one alternative representation for performing the operations presented in this chapter.

Actor machines are also used to classify actors as deterministic, prefix monotonic, Kahn processes, CSDF, and SDF [34]. This analysis could easily be extended to identify the simplified schedulers presented in this chapter, with the added value of a more efficient description of the action selection process providing information regarding which values are known at which states.

Chapter 5

Analysis of Actor Partitions

Scheduling on partition level is concerned with deciding on an interleaving of concurrent tasks, respecting data and control dependencies. In a dataflow program, data dependencies are explicitly described by production and consumption rates, and define when a task becomes enabled for execution. Control dependencies on the other hand, describes what functionality, or which task, becomes enabled.

Figure 5.1 shows an example of the scheduling operation. The state machine describes the different program states the partition can reach between action firings, and, for all possible firing sequences, two schedules (black) are chosen while the other states (gray) are removed as these will never be used. The state, σ_1 is a state where the guard depends on input and for this reason, a choice between the two paths are to be taken at run-time. The other states that are included in the scheduling has only one outgoing transition which means that there is no scheduling decision. In practice, this means that a scheduler with two schedules, each consisting of four transitions (actions) can be generated for this partition. This scheduler can

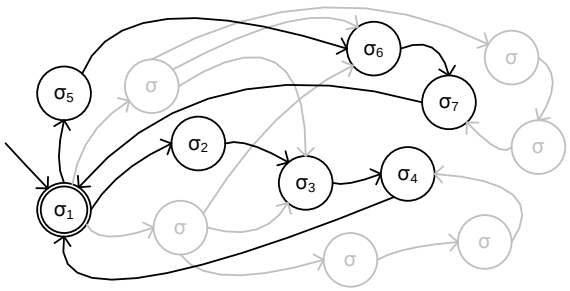

Figure 5.1: Scheduling of a partition is about choosing a minimal number of paths in the program state space, such that is still can process the full set of inputs.

then be described as an FSM with one state corresponding to σ_1 and two transitions corresponding to the two sequences of transitions in the original FSM.

For the new scheduler to be valid it must be shown to accept all the possible inputs accepted by the original program and it must be deadlock free. There are in principle three types of behavior that affect these properties: 1) different types of inputs which affects the scheduling, 2) variable token rates which cannot be decided from the partition, and 3) feedback loops outside the partition. The scheduling process makes the scheduling more static, which means that the partition does not react at each individual input but waits for more inputs which correspond to the inputs of the full schedule. If this is not handled properly, and the scheduler becomes more static than the input stream to the partition, the result is a scheduler that may deadlock.

The actual scheduling is done by analyzing the state space of the program. This model must describe data rates, control values, any scheduling decision performed by an actor, and the communication queues. For the state space analysis to be feasible, any information not related to scheduling is removed from the model. This raises the question what the minimal information needed, that still describe the complete behavior of the model, is. To show that the scheduling model is correct, the behavior of the model must correspond to the behavior of the program, the input sequences of the model must describe every possible input sequence, and the current state of the model must be described such that everything related to scheduling, but nothing else, is described.

In this section the scheduling model, how it is constructed and how it is shown to be complete is discussed. After this the actual scheduling procedure is described, followed by a discussion of some related approaches.

5.1 The Scheduling Model Format

The Promela language (Protocol Meta Language) is used to generate the model for the state space analysis. Promela is based on Communicating Sequential Processes, and can easily be used to describe the behavior of a CAL program. The Promela language is used as the input to the Spin model checker [72] which is used to verify some properties of the model, but also to find the actual schedules.

A CAL actor loosely corresponds to a Promela Process, with the difference that the CAL actor has the notion of firings which needs to be modeled in Promela. For this, a CAL action can in Promela be represented by a guarded command where the action body is encapsulated inside an *atomic* statement, as follows.

```
proctype actor1 {
  int x, y;
  do
  ::  atomic { "guard expression" -> "action body"}
  ::  atomic { "guard expression" -> "action body"}
  od
}
```

This actor has two actions which are repeatedly fired according to the guards and availability if data. The action inside the *atomic* statement is executed if the guard expression evaluates to *true*, after which the action is executed without interruptions from other processes. The execution of the Promela program is therefore sequential and the smallest executable unit is the atomic statements corresponding to CAL actions, and thus, the notion of action firing is added to the model [1]. CAL constructs, such as the FSM scheduler, are simply implemented as variables and extensions to the guard expressions, which means that the FSM state of the actor simply is described by an integer value while the different states are described as an enumeration.

```
proctype actor2 {
  int x, y;
  do
  ::  fsm_state==state_1
       if
       :: atomic { "guard" -> "action body"; "update FSM"}
       :: atomic { "guard" -> "action body"; "update FSM"}
       fi
  ::  fsm_state==state_2
       if
       :: atomic { "guard" -> "action body"; "update FSM"}
       :: atomic { "guard" -> "action body"; "update FSM"}
       fi
  od
}
```

The Promela *actions* are further instrumented to be useful for the scheduling by adding code for tracking which actions have fired and whether or not a program has fired actions at all. Also, state variables which are relevant for scheduling and the variables representing the FSM, are made global and accessible from any expression regarding the global state of the program that is needed for the state space analysis. To avoid problems with naming clashes, the variables are renamed with a prefix corresponding to the name of the containing actor.

[1] It is worth to note that it is essential that the guard is strong enough regarding queues as a blocking queue brakes the atomic statement.

The translation of CAL actors to Promela processes is quite straight forward, conceptually there are no difficulties and as long as the code generation is correctly implemented, the model of the actors will be correct. What is more interesting in this context is how to describe input sequences that fully exercise the program as well as choosing a set of guards that cover the possible input sequences. Further, the program should be partitioned such that strong enough guards can be found without the need for complex composed guards, as described in Chapter 3.

5.2 Correctness of Scheduling Model

The Promela model is a replica on the original CAL program, and with a given input, it will fire the same actions as the CAL program is specified to fire. The outputs of the program, however, will not be correct as the Promela model only contains the instructions that are relevant for scheduling. This means that for any given input, the Promela model can be run, and is guaranteed to fire actions according to the CAL program, but keeping data as valueless tokens while control information exactly corresponds to the CAL program.

A separation of control and data tokens and variables is necessary, not only for avoiding state space explosion, but also for identifying which variables to monitor during the analysis, and for generating the set of input sequences that the program should accept. For the inputs, this is essential, as the sequences should be described as the few possible control values plus streams of valueless data tokens. The implication for the model is that the control tokens and the guards in the partitions should be a perfect match, not only for the guards directly testing the inputs but also any guard that in any way may be affected by these values.

Control Variables and Channels To reason about the state of a program partition, a minimal set of variables, queue states, and FSM states, that completely describe the state of the partition from a scheduling perspective need to be identified. The Promela program is not fed with real inputs, instead, a set of input sequences that represent every possible input is generated and schedules represent processing each of these sequences.

The scheduling model describes the possible input sequences S (see Figure 5.2) which can be considered to describe the different types of *data* that can enter the partition. The state of a partition is described by the state variables in V_S, the states of the actor FSMs, and the states of the FIFOs inside the partitions. The scheduling states $\Sigma_S = \{\sigma_1, \sigma_2, ..., \sigma_M\} \subset \Sigma$ is an as small as possible subset of the set of possible program states, where each state represents the state reached after processing an input sequence start-

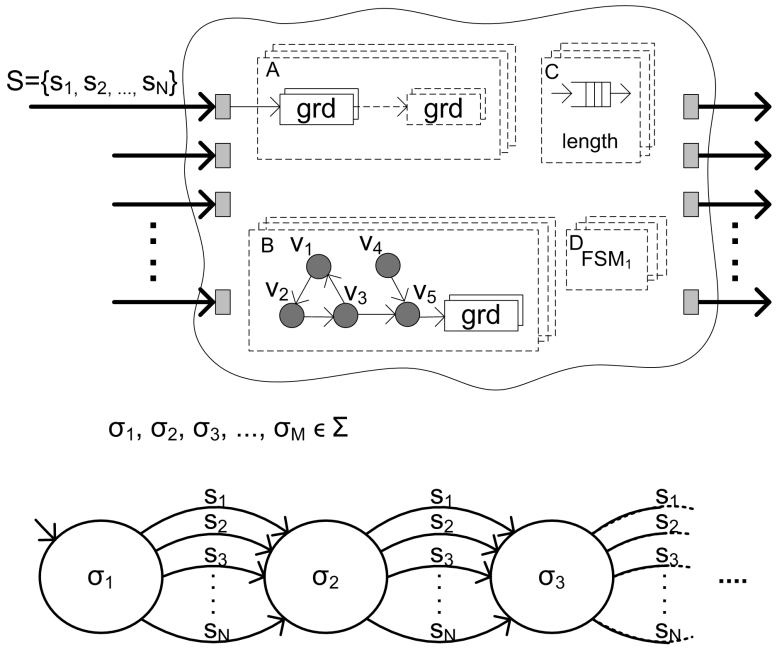

Figure 5.2: The scheduling model from the point of view of control tokens and variables.

ing from a previous state in Σ_S. The generated scheduler is a state machine where the states correspond to the program states Σ_S and the transitions correspond to consuming one of the input sequences in S by firing a sequence of actions i.e. a schedule.

What is meant with correctness of the scheduling model is not, by itself, obvious and should be defined more precisely. The post scheduling program may have token rates different than the original program and some behavior of the original program may not be available in the transformed program. Now, this does not necessarily indicate incorrect behavior, but it requires some attention as it is possible to introduce deadlocks or unwanted behavior. There are two aspects regarding a scheduling model for actor composition that are relevant in this context, the first has to do with describing the input sequences of the partition and the second with what can, or needs to be, guaranteed about potential nondeterminism in the model.

The dependency graph presented in Chapter 3 represents, in a simple fashion, the control information paths in the partition. Every variable in the graph is part of describing the program state; furthermore, the graph also describes where control information enters a partition and how it propagates to the actors in the partition. For the Promela model to be correct, the produced schedules must completely represent the operations of the original program. The schedules represent processing an input sequence and the

choice of schedules is to be made by checking the type of the input and the state of the program. For this reason, a set of guards that are strong enough must be identified.

Identify the Guards and Input Sequences For an input sequence, the guards that check a property of the input may be located in more than one of the actors of the partition. In practice, this means that the scheduling is distributed and decisions are made at different times during the processing of an input sequence. To be able to choose a static schedule based on the data waiting at the input ports of the partition, a set of guards covering all these other guards is needed. This can be achieved either by choosing proper partitions or by composing a set of stronger guards from the guards of the partition.

The relationship between input sequences, guards, and the scheduler is illustrated in Figure 5.2. The section of the partition description labeled A, shows a set of guards directly reading the input sequence and another set of guards (connected by a dashed lines) which at some later point checks some property of the input. This latter set of actions should be covered by the guards in the first set, such that evaluating the guards checking the input provides the information that predicts the evaluation of the other guards.

In this work, the preferred approach is to choose partitions such that there is one actor with strong enough guards to schedule the rest of the partition. This approach restricts the sizes of partitions as some actors have guards that are not compatible, however, it also guaranties that the scheduling overhead is never increased as guards are removed but never added. Further, it also enforces a partitioning where the processing is related to the same properties of the data as the guards must be overlapping. Of course, there will certainly be cases where creating new guards will give some benefits and it would make sense to explore this optimization problem, but it is left outside the scope of this thesis.

The input sequences for the actors in a partition are generated according to the actor level scheduling presented in the previous chapter. Control values are resolved using the SMT solver approach presented in [124] while data values simply are a number of valueless (in practice zero value) tokens. When these are put on the input channels to a partition, the model checker is ready to run as soon as it has been verified that the guards are strong enough to identify every different input sequence that requires a different schedule.

Verify Guard Strength The guard strengths are analyzed according to the types of control value propagation presented in Section 3.2.2. In the cases where the propagated control value is not a function of the input, the

one called *galvanic isolation*, no further verification is needed. For the other types, where two actors make scheduling decisions depending on the input sequence, the chosen guards must be shown to be strong enough.

In the case called *shared property*, where the control is produced as from a pure function by not keeping the input in state variables between action firings, the verification of the guards is reasonably simple as it is not necessary to analyze longer sequences of inputs as in the case with the *shared information path* type.

When the guards does not depend on variables with history of inputs, a verification of every pair of inputs is enough; possibly combined with the possible values of any variable used for generating the control values. This verification can be done with a model checker as the state space does not grow too large, however, it is essential that control tokens are represented with no more bits than necessary as every bit enlarges the state space enormously. Below parts of a Promela program used for verifying the guard strengths for a partition with two actors is presented.

Consider a CAL program with two actors, where the first actor fires one of two possible actions depending on a control value that is read from an input port. Both of these actions produce a control token which is used by an other actor which reads this value and in turn fires one of two actions based on this value. The question to be answered is whether the guards of the first actor can be used to predict which action will fire in the second actor. The first actor is fed with two values from the set of numbers corresponding to the bit-length of the control input. The select statement chooses an arbitrary number from the set 0..4095, and when running model checking, each of these are checked. This value is then pushed to the input queue of the first actor (chan_IN); by pushing two values to the queue, the result of all combinations of the control input values are checked.

```
control=select (i : 0..4095);
chan_IN!control;
control=select (i : 0..4095);
chan_IN!control;
```

The actors are modeled simply as the guards, a counter of how many times the action has fired, and possibly the code that generates control output. In this example the guards of the actor called *actor1* are to be used as the guards in the scheduler. The functions *grd_a1* and *grd_a2* represent the guards of the two actions of *actor1*, the actual guard expressions are not important at this point. In a similar fashion, the functions *f_a1* and *f_a2* represent the code that produces the control output that is used in the guards of the other actor.

```
int d_a1=0;
int d_a2=0;

proctype actor1() {
int control;

do
:: skip -> chan_IN?control;
   if
   :: grd_a1(control) -> d_a1++; chan_MID!f_a1(control);
   :: grd_a2(control) -> d_a2++; chan_MID!f_a2(control);
   fi;
od;
```

The second actor is modeled in a similar fashion. The two actors are connected with a FIFO channel named chan_MID, which transports the control value between the actors.

```
int d_a3=0;
int d_a4=0;

proctype actor2() {
int control;

do
:: skip -> chan_MID?control;
   if
   :: grd_a3(control) -> d_a3++;
   :: grd_a4(control) -> d_a4++;
   fi;
od;
```

The verification is simply performed by adding a set of assertions that must hold after the program has processed the two input types. Every pair of input values, where both tokens are accepted by the same guard in *actor1* and thus the first part if the expression is false, then either of the following Boolean expression must be true. In other words, if *d_a1* did accept both tokens, then either *d_a3* or *d_a4* must accept both tokens generated by *actor1*. This corresponds to the requirement presented in formula (3.1), defining when actions can be considered to have a functional dependency.

```
timeout;
assert (d_a1 != 2 || d_a3 == 2 || d_a4 == 2);
assert (d_a2 != 2 || d_a3 == 2 || d_a4 == 2);
```

The above model has unnecessarily generic input as it is only generated based on what is known from the actor that reads the input. Using the

actor that produces the control tokens may potentially allow a more exact generation of the input values that need to be checked, in some cases some values that in practice can never occur may result in that the verification is not successful and it is then necessary to describe the input more carefully.

Model checking and an exhaustive state space search is one way to prove that the guards are compatible, it is however unnecessarily complex and does not scale well with the size of the input. Alternatively a theorem prover can be used to prove this property. The model checking approach was used in this work as the Promela representation of actors was available. To demonstrate alternative methods, the example in Figure 5.3 shows how the corresponding verification can be done in Rodin [5].

```
AXIOMS
 axm1:   T ⊆ ℕ
 axm3:   grd1 = {x | x ∈ T ∧ x < 10}
 axm4:   grd2 = {x | x ∈ T ∧ x ≥ 10}
 axm7:   out1 = {x | x ∈ T ∧ (∃ y · y ∈ grd1 ∧ x = y)}
 axm8:   out2 = {x | x ∈ T ∧ (∃ y · y ∈ grd2 ∧ x = y)}
 axm5:   grd3 = {x | x ∈ T ∧ x < 10 ∧ x ∈ T}
 axm6:   grd4 = {x | x ∈ T ∧ x ≥ 10 ∧ x ∈ T}

 thm1:   ∀ a, b · a ∈ out1 ∧ b ∈ out1 ⇒
            (a ∈ grd3 ∧ b ∈ grd3) ∨ (a ∈ grd4 ∧ b ∈ grd4)

 thm2:   ∀ a, b · a ∈ out2 ∧ b ∈ out2 ⇒
            (a ∈ grd3 ∧ b ∈ grd3) ∨ (a ∈ grd4 ∧ b ∈ grd4)
END
```

Figure 5.3: Variable strength test in Rodin. The two theorems are simple enough to be automatically proven.

In the example, *grd1* and *grd2* represent the input tokens accepted by the guards of the first actor, *out1* and *out2* represent the outputs possible from these two actions, and *grd3* and *grd4* represent the guards checking those values. The guard in this example simply check if the input is larger than ten or not and the actions simply resends the input value as the control output value. The theorems state that any two values accepted by the same guard in the first actor must also be accepted strictly by one of the guards in the second actor. With the theorem prover, this property is instantly and automatically proven. Compared to the model checking technique, the theorem prover does not suffer from state space explosion which means that this kind of problem scales well.

When the guard strength can be verified, the guards of this one actor are the guards that are needed to schedule the partition. If it cannot be verified, either the guards has to be made stronger or a different partitioning is needed.

Variable Token Rates Finally, to show that the scheduling model is complete, it is essential to show that the token rates of the different inputs are synchronized. Following the approach presented above, one actor is chosen for providing the guards of the partition scheduler. In a similar fashion, this same actor should also decide on the consumption and production rates of the other actors in the partition. This means that the other actors should not be allowed to make a choice depending on input, be it based on input value or input availability, except if that choice can be decided from the actor on which the scheduling is based.

In a partition, therefore, there should only be one actor which can make independent scheduling decisions based on the available input. All the other actors must depend on this choice and follow the first actor.

This property must be identified for a partition. By showing that one actor has enough strong guards to schedule the other actors, this also means that there are no independent scheduling decisions based on input values in the partition. The other type of independency comes from actors that have nondeterministic or time-dependent behavior. The classification presented by Wipliez et al. in [124] can be used to identify such actors, and an actor that is classified to belong to KPN or a more restricted MoC is known to not have such a behavior. However, an actor with this type of behavior can still be allowed to be part of a partition as long as it can be shown that this behavior is only available inside the partition but not available on the input ports of the partition. The simple rule, therefore, is that all input ports to the partition should belong to actors which completely depend on the actor which the scheduling is based on.

In principle, there is no limitation for the actor which the scheduling is based on to have nondeterministic of time-dependent behavior, this is simply a scheduling decision as any other scheduling decision. However, the goal with the scheduling is to work on larger units of input, this again may be in conflict with basing the scheduling on a time-dependent actor as the amount of input then affects the scheduling. For this reason, it is avoided to have time-dependent actors connected to the input ports of the partition. Now, when this actor which scheduling decisions decide the token rates and behavior of the other actors in the partition has been identified, the next step is to search for the actual schedules of the partition.

5.3 Schedule Construction

The actual scheduling is a composition of the actor state machines which in this case are represented by Promela processes. The composed scheduler describes a set of reoccurring program states which are linked by a set of schedules, representing processing the possible inputs of the program. This

means that the composed scheduler typically is less general than a parallel composition of the actors, where the behavior of the program would be exactly the same as in the original program. The new scheduler will require more data to be available before the processing can start, however, for a specific input, the output is guaranteed to be the same as for the original program, but slightly delayed.

The practical scheduling consists of identifying the schedules to search for, instrumenting the Promela program to correspond to the state where the schedule starts, describe where the schedule should end, and analyze the model checker outputs.

Identify Schedules The schedules to search for correspond to the guards which have been verified to cover the possible input sequences for the partition. When the guards are the guards that checks properties of the inputs of one actor in the partition, the scheduler of the partition can be assumed to resemble the scheduler of this actor. For this reason, one of the actors of the partition that has the guards that completely covers the inputs is chosen as the partition *leader* and is used to generate the initial scheduler for the partition.

The initial set of schedules will for this reason correspond the actor level scheduler, of the leader actor, as described in the previous chapter. A schedule then corresponds to the sequence of actions that can fire in that actor without evaluating any guards, furthermore, the schedule of the partition also includes the actions that fires as a consequence of the action sequence that fired in the leader actor. The state machine of the generated scheduler corresponds to the states of the FSM of this actor where an input dependent condition must be evaluated in order to choose the next action to fire.

The description of the schedules to be found consists of three parts, the FSM state of the leader actor, the actual state including FIFOs, variables and FSM states of the whole partition, and the action from which the guard that enables the schedule is taken. The starting point of the scheduling may look like the following example.

```
[0] inter : read -> read : (s0 -> ?)
[1] intra : read -> read : (s0 -> ?)
[2] other : read -> read : (s0 -> ?)
[3] start : read -> read : (s0 -> ?)
```

These four specifications represent four schedules, which, at least, are needed for scheduling the intra coding part of a MPEG-4 decoder. Each schedule is represented by the enabling action, the start and end state of the leader action and finally the start and end state of the partition. In this

Algorithm 5.1 Scheduling Process – Constructing the composed FSM by finding schedules corresponding to the schedules of the leader actor; when a new state is found, the schedules starting at the corresponding FSM state of the leader actor ($\sigma_{leader} \approx \sigma_{new}$) are also searched for.

1: $\Sigma_{partition} := \{\sigma_{init}\}$ ▷ Concrete states
2: $S_l := \{(\sigma_{start}, \sigma_{end}), ...\}$ ▷ Schedules of the leader actor
3: $S_m := \sigma_{init} \times \{k \mid k \in S_l \wedge dom(k) \approx \sigma_{init}\}$ ▷ Schedules to be found
4: **while** $S_m \neq \emptyset$ **do**
5: $s :\in S_m$
6: $(\sigma_n, t) := findschedule(s)$ ▷ Run model checker
7: $S_m := S_m \setminus s$
8: **if** $\sigma_n \notin \Sigma_{partition}$ **then** ▷ New state found
9: $\Sigma_{partition} := \Sigma_{partition} \cup \{\sigma_n\}$
10: $S_m := S_m \cup \sigma_n \times \{k \mid k \in S_l \wedge dom(k) \approx \sigma_n\}$
11: **end if**
12: **end while**

case, the action, or the guard of the action, checks if the input is an *intra* or *inter* block, if the input is coded without intra prediction, or if it is the beginning of a new frame. The next field states that the schedule should start and end at a state where the leader actor is in the state *read*, and finally the last field states that the schedule starts at a state *s0* and ends at a yet unknown state. The difference between the states *read* and *s0* is that the first indicates any state where the leader actor is in its FSM state called *read* while the second state describes the full state including state variables and the FSM state of each of the actors.

The last part of the scheduling description is actually the most interesting. The scheduling step is completed when each of the question marks has been replaced with a concrete state. When a schedule is found, if the schedule ends in an already known state, the question mark is simply replaced with this state, if the state is not known from before, a new state is added to the scheduler. A new state means that it is necessary to find schedules leaving that state too. As an example, when a schedule is found for the intra schedule in the previous description, it does not correspond to the *s0* state, which means that a new state, *s1*, is added to the scheduling description. The new schedules needed from *s1* are those representing the actions that may be enabled in the state *read* of the leader actor as *s1* corresponds to the state *read* of the leader actor according to the specification. The scheduling process is described in Algorithm 5.1, which shows how the schedules of the leader actor and the initial state of the partition are used to produce the appropriate scheduling tasks for the model checker to find.

Finally a complete scheduler is found for the partition, which may look

like the following. Here the scheduler takes into account every input sequence that can occur at the same time as it considers the different states the partition may end up in after processing the different input sequences.

```
[0] inter : read -> read : (s0 -> s1)
[1] intra : read -> read : (s0 -> s1)
[2] other : read -> read : (s0 -> s0)
[3] start : read -> read : (s0 -> s0)
[4] inter : read -> read : (s1 -> s1)
[5] intra : read -> read : (s1 -> s1)
[6] other : read -> read : (s1 -> s0)
[7] start : read -> read : (s1 -> s0)
```

In practice, the concrete state of the partition must be used to instrument the model checker and be compared to the states the model checker finds when it performs the state space analysis.

Instrumentation of Model Checker Describing the state of the program partition as well as configuring the partition to a specific state is essential for constructing the schedules for the partition. Three operations are needed for the model checking. First, we need to describe the initial state of the program such that the scheduling information is exactly described while data properties are skipped. Second, we need to read the exact state of the program when a schedule has been found. And, third, we need to be able to setup the program to each of the states that has been identified.

A state of the Promela program can be found either by simulating the program and reading the end state or by running model checking and reading the end state of the checking in the case a state that was specified to be searched for was found. To find the initial state of the program, a simulation run where no action is allowed to fire is run, as variables not related to scheduling already has been removed from the model, capturing the state variables of the partition gives the exact initial state of the Promela program. Then from this forward, the program is always initialized with either the initial state or one of the other found end states.

```
init {
  #include "state.pml"
  atomic{
  run actor1();
  run actor2();
  ...
  }
}
#include "tmp_ltl_expr.pml"
```

Search for Schedules The last piece of the model checking puzzle is the description of the state to look for. Now, model checkers are typically meant for verifying correctness of programs, thus correctness properties that should hold for the program are described and errors are reported by the model checker when a violation is found. Typically an error state or a sequence of states that should never occur are described with *linear-time temporal logic* which enables reasoning about the order in which things happen.

In the search for schedules, we are interested in finding specific states in the program state space and the sequence of action firings that leads to that state from the state to which the model checker was initialized. To make the model checker to search for a specific state and report that the state is reachable providing the path (the trace) to this state, the state needs to be described as an unwanted error state. This kind of (miss-) using the model checker is needed when a general model checker that is designed to verify correctness and not schedule existence is used for this purpose.

The state to be found can simply be described as a *linear-time temporal logic* formula, as is shown in Formula 5.1, which simply says that for every state (the square), there should not be a state where the program has made progress, by firing at-least one action, and where the FSM of $actor_1$ is in the state named $state_1$, and the data of the FIFOs of the partition has been consumed. What is actually described here is that, first of all, a schedule requires that something has been done, else the schedule is useless. Second, a partial specification of the state to be found is described typically describing how one of the actors should be run during this schedule, and last, some description of how the other actors have synchronized with the tokens that has been produced by getting to the specified state.

$$\text{ltl } \Box! \left(hasProgress \wedge FSM_{actor1} == state_1 \wedge emptyBuffers \right) \tag{5.1}$$

The spin model checker generates an automaton, based on this formula, that is executed in lock-steps with the program that is model checked. The automaton describes a sequence that should never occur and it is an error when the automaton terminates; the generated trace to the error can then be used in combination with the simulator to run the program according to the schedule that ends up in the specified state [72]. The simulator can then be instrumented to generate the information that is needed to both get the schedule and the state that is needed to decide on how the new scheduler needs to be updated to correspond to the information gained from running the model checker.

Model Checking Results The output from the model checker, when the described state is found, is the sequence of action firings that lead to that

state and the complete state description of the state that corresponded to the described state. The sequence directly corresponds to a schedule while the state needs to be compared to the other states already found and decisions about how to update the scheduler are made from this comparison. The complete state description is a listing of the state variables that are used in scheduling, the contents of the FIFOs, and the FSM states represented by variables in the Promela representation.

```
//s0
var_actor1_var1=VAL;
var_actor1_var2=VAL;
fsm_actor1=state1;
```

The resulting state, after the schedule search, is compared to already reached states to find a minimal number of states that can describe the new scheduler. The state variables that are used to describe the state are only those that are reachable from a guard in the dependency graph, it would else not be possible to generate a neat scheduler if also data variables would be included as the number of states would become enormous. To further improve the precision of the state description, a state variable filter that describes, for each FSM state, which state variables are relevant, is used. This filter is based on the analysis described in the previous chapter (in Section 4.5), and it simply keeps track of, for each state, whether a variable will be overwritten before being used in a guard expression. In some cases, as will be demonstrated in *Case Study V* in Chapter7, this can significantly reduce the number of states in the generated scheduler.

The reached states are used when the Promela representation is instrumented for a scheduling run. The state that is included in the model looks like the example below, containing the assignment of the state variables, FSMs, and pushing the relevant inputs to the input channels of the partition.

```
//s0 state
var_actor1_var1=VAL;
var_actor1_var2=VAL;
fsm_actor1=state1;
//inputs
chan_actor1_port1!CNTL_VAL1;
chan_actor1_port2!0;
...
```

This completes the scheduling cycle. The model checking gives a more and more exact description of the scheduler needed; this in turn is used as input to the model checker. Finally when each reached state has an outgoing

transition (schedule) for each type of input that is accepted in that state, the scheduling is completed.

5.4 Scheduling Strategies

With the vast number of behaviors that can be implemented in a dataflow network, it is no surprise that one scheduling strategy will not be sufficient for every program. Luckily, a few variation of the one strategy presented above, covers the majority of programs. The key point is to decide which the states to search for are, as the other parts of the scheduling is general and corresponds to the behavior of the dataflow program. The description of the end state of a schedule, however, is either a state that can be reached or not. The problem then is, how to describe a state which is reachable, but at the same time results in a proper scheduler.

The initial scheduler is often given by one of the actors' simplified FSMs, from which the input sequences that should the processed by the schedules are defined. When there are no data dependences, the choice of initial scheduler is arbitrary and could be omitted, and a schedule could correspond to processing *all the available data*. In general, what can be used to describe the state of a partition when searching for a schedule is: FSM states of actors, state variables of actors, FIFO channel contents, and progress of the actors. The problem is that, if the state is described to exactly, it may not be found, and, if it is described to loosely, the scheduler will probably have many unnecessary intermediate states. The only property that is always required is progress as, a schedule does not make sense if it does not fire any actions, however, different trade-offs between the other properties are needed.

One FSM Plus Empty Buffers A strategy that works for many actor partitions is to define a schedule to be finished when every internal FIFO of the partition is empty and the inputs of the *leader* actor have been consumed. The state description then contains the FSM state of one of the actors, the requirement that the FIFOs are empty, and of course the requirement of progress; this is the strategy presented throughout this chapter. The state variables are completely outside the description, however, they are considered when constructing the scheduler. The requirement of having every FIFO empty between the schedules is not always possible, and a variation of the strategy is needed to handle these cases.

Allowing Delay Tokens Programs with feedback loops may often be implemented to, once initialized, never reach a state with empty buffers. A simple solution to this problem is to simply remove some of the FIFOs from

the requirement of emptiness, and by this allow the program to *leave* some (valueless data) tokens in the partition between the schedules. These tokens now need to be part of the state description, and need to be replaced when the program is initialized to that state the next time.

There are a couple of alternatives for how to choose which FIFOs to allow to keep tokes, based on how delay tokes usually are used. Often delay tokens are needed together with feedback loops, which makes the feedback FIFOs a natural position where to leave the delay tokes. Another possibility is to consider how delay tokens are generated in CAL. A delay token is generated once, meaning that there probably is an actor which has an action that can only run once, after which the actor continues in its normal execution loop. These actors can be identified, and the potential FIFO for keeping the delay token is the one where it initially is produced. However, identification of delay tokens is outside the scope if this work and is instead left for the developer to decide on and inform the scheduling framework.

All FSMs Specified Sometimes a specific strategy can be used to force the scheduler to use an already known state, if it is reachable. This is useful when the partition contains actions which does not affect the state of the FIFOs inside the partition, typically by having an action that changes the FSM state of an actor but does not communicate with ports. In this case, the state that is searched for can be described more exactly by specifying the required states of each FSM according to an already known state. By doing this, the final scheduler becomes smaller.

5.5 Actor Composition

Schedules are created to be executed. A schedule can be seen as a list of actions that will fire in sequence and can be glued in to the C program generated from the CAL program as a list of function calls to functions representing the actions. It is, of course, more practical to be able to transform the CAL program according to the new scheduler and allow code generation from the transformed CAL program. The scheduler created by the approach presented in this chapter consists of an FSM scheduler, a set of guards based on the guards of the original actors, and sequences of action firings representing transitions of the FSM. This obviously resembles a CAL actor, and can be seen as a composition of the original actors.

In practice, the FSM from the generated scheduler becomes the FSM of the composed actor, the schedules become actions, the needed guards are added to the actions, and the token rates are calculated from the schedules. With actor composition, the composed actor is more predictable than the set of original actors. The communication between the actors is known at

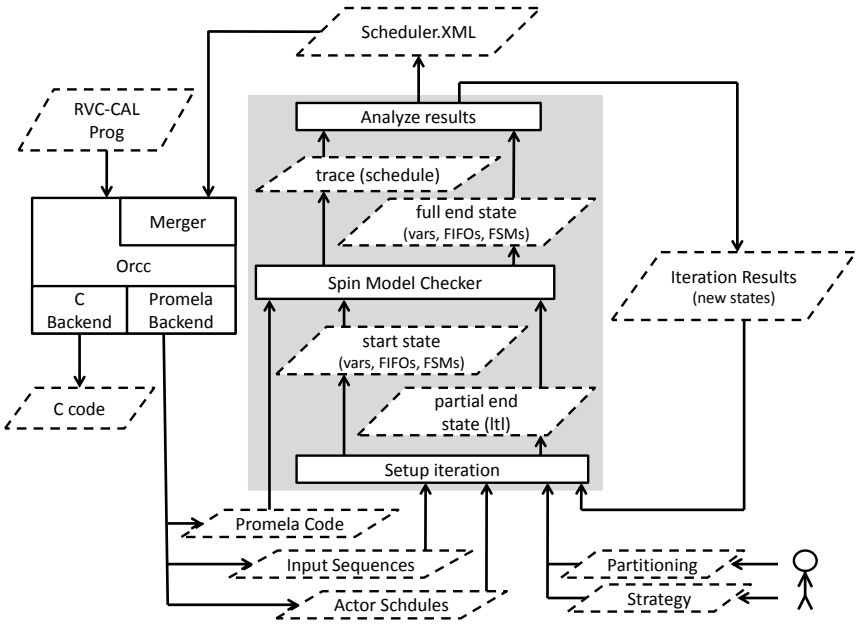

Figure 5.4: The scheduling tool chain, shown as the gray area, and its connections to the other tools. The rectangles represent different software artifacts while the dashed shapes represent different pieces of information that is transfered between the tools.

compile time which means that the FIFOs can be replaced by variables and there is no need to perform any checks in these intermediate buffers. The compiler is also provided larger pieces of predicable sequential code that it can optimize.

The construction of a merged actor used for the output from the model checker scheduler, is based on the actor composition presented in Paper 5, which by now has been implemented [2] in the Orcc compiler as an actor merger which constructs an actor composition based on a scheduler specification specified in an XML format.

The full tool chain is shown in Figure 5.4, where on one side the Orcc compiler is responsible for producing both code for the actual executable program and for the model checker including the specification the model checker needs regarding the analysis of the individual actors. The merging tool is also part of the Orcc compiler and receives the composed scheduler from the scheduling tool. On the other side of the figure, named *scheduling tool*, we have the CAL program translated into Promela code, a set of scripts that instruments and runs the model checker, and a set of XML files

[2]By Jani Boutellier and Ghislain Roquier.

including partial schedules, scheduling information about individual actors, and information about input sequences.

5.6 Related Work

Scheduling of dataflow programs is far from a new problem and it has been investigated for several decades. The scheduling problem can be divided in to two categories, 1) designing dataflow programs where actors have restricted behavior according to a MoC which guaranties a certain level of predictability, or 2) designing the program with a very expressive language and attempt to fit parts of it in to more restricted MoCs. For the first approach many MoCs with a different trade-off between predictability and expressiveness has been proposed and several of these were presented in Chapter 2. For the second, the scheduling can be viewed as fitting actors to more restricted models which can be statically scheduled, as presented in the last chapter, or by scheduling a part (or partition) of a program as presented in this chapter. There are several approaches that are relevant to be compared to the scheduling approach presented in this chapter.

Dynamic Code Analysis The scheduling approach that resembles the approach presented in this chapter the most is dynamic analysis presented by Boutellier et al. in [28]. Here, the executable code generated from the CAL code is instrumented to collect the needed information from the program while the program is executing. The approach is called dynamic as the information is collected during program runs with real input; this means that the program is analyzed during normal execution, however, it also means that the inputs must cover all types of possible inputs for the generated schedules to make sense.

A quasi-static scheduled actor network is defined as a set of actors where a control value entering the partition on a FIFO, decides which operating mode (or schedule) to be fired. This definition matches the behavior of PSDF [20], by seeing the control token as the parameter that controls the operation of the partition. This also, of course, requires that the control token has total control over the actor partition [28]. This is similar to the restrictions we have on control tokens in the approach presented in this thesis, the difference is how this property is verified. Compared to the analytic approach presented earlier in this chapter, in [28], the program is instrumented with *token gates*, which are added to FIFOs that carry control information. A *token gate* is a construct that has two objectives: first, it controls when the control tokens can enter the network and by this can delay the next control token until all the computations related to the previous one has finished, and second, it observes the control tokens passing through in

order to use these as parameters [28].

The dynamic analysis then collects, for each actor, action chains, which consist of a sequence of action firings and a corresponding *signature* which describes the parameters that made the actor run that specific sequence. Further, on the network level, the values of the control tokens are mapped to sequences of actors that are invoked for that specific value. These sequences are called strands, where a slot corresponds to an actor and can contain a number of action chains. The result is then a set of actor strands corresponding to control values and a set of action chains corresponding to the set of signatures for the actors. [28]

Boutellier et al. presented a further improved version of this approach in [27], with improvements mainly regarding the segmentation and the generation of quasi-static schedules. We can for the purposes of this chapter see the production of action chains to be similar to the one presented in [28] as it is also discussed in Chapter 4 related to the actor level scheduling. The concepts of *actor strands* and *token gates*, are however dropped here and the network level scheduling is made more general by constructing the segments according to a set of rules regarding the behavior of the actor and the scheduling is constructed by expanding the concept of signatures to the level of segments.

The approach in [28] has to some extent been used as inspiration when developing the approach presented in this thesis. The model checking approach has some advantages over the dynamic code analysis. First, with a model checker, it is possible to search for a specific state, while when running a program, it is, in some cases, difficult to accidentally get the program to interleave the action such that the wanted state is reached. Second, the inputs should be proved to be complete, how this can be done has been demonstrated in this chapter, and could probably be extended to the approach in [28, 27].

Statically Schedulable Regions Another approach, presented by Gu et al. [60] identifies *SDF like* regions in CAL programs which can be scheduled statically. The end result is a quasi-static scheduler, where groups of computations across several actors can be statically scheduled while these groups are dynamically scheduled. The idea is to identify statically schedulable regions, which can be seen as a set of ports from several actors, which are connected by one of several possible relations. The identification of statically schedulable regions mainly consists of constructing a graph describing the association between ports after which various transforms are performed on the graph.

The initial partitioning of ports, which is performed per actor, is based on the interaction between the ports. If two ports either are used in the same action, or one port is used in a state changing-action while the other

port is used in a state-guarded action, then the two ports are in the same partition and are called *coupled ports*. Then, a *coupling relationship graph* is constructed by extending the partitioning to span over several actors by using the information about how the queues connect the ports in the CAL program. The partitioning so far describes how the ports directly depends on each other; to make something useful out of this, the next step is to identify *coupled groups*, which are the weakly connected components of the *coupling relationship graph*.

The next step is to extract from each *coupled group* a subset of ports into *statically related groups*. These ports are *static rate ports*, which means that each action using that port has the same rate for that port. Furthermore, the subset of ports, must be such that any action of the actor, either has each of these ports or none. This means that, for each action firing of that actor, either, each of these ports will consume/produce a static number of tokens, or nothing. This property is related to the ports, the actions using these ports can still include dynamic ports, however, these ports have a static relationship regarding token rates.

By connecting the *statically related groups* according to the CAL network, and by this constructing a graph, *statically schedulable regions* can be constructed by taking the *weakly connected components* form this graph. The *statically schedulable regions* are then a set of ports across the dataflow network which acts like an SDF graph. Due to the partitioning, these are also independent, as ports related by state modification are in the same partition or outside the partitions. This means that the static regions can be extracted as separate actors, where the ports connecting the *statically related groups* disappear inside the actor, while the other ports remain as the ports of the new actor.

This work is quite different from the presented approach which searches for schedules by analyzing and identifying the dynamic behavior, while this approach identifies the static behavior. It is not completely clear how the approaches could be used together, one possibility is to first identify *statically schedulable regions* and extract these as separate actors, then use the presented approach to schedule the dynamic parts possibly including the static regions.

Analysis of Actors with SMT The actor classification approach, which was already discussed in the previous chapter, presented by Wipliez et al. [124], involves analyzing the guards of each actor with an SMT solver (Satisfiability Modulo Theories). This involves finding the *parameter* that exclusively satisfies the input dependent guards when an actor is classified as quasi-static, but also to identify time-dependent behavior. For the discussion in this chapter, the approach for comparing whether guards are compatible or not, is interesting as is relates to the problem of checking the

guard strengths. It would make sense to investigate whether an extended version of this work could be used to check the compatibility between guards of separate actors, that is, to find if the guards of one actor are strong enough for scheduling a partition. Furthermore, parts of this work are directly used for generating the model checker information.

In generating the Promela program as presented above, the guard solver is directly reused to produce the concrete control tokens. This is only available for control tokens that are peeked, so far, but could be extended to allow other types of input values; the problem being that if the control value is modified before reaching the guard, the analysis becomes more complex, and if the guard does not purely depend on that input, it may be impossible. Also, the functionality to identify if an actor has time-dependent behavior is used to find the actors that must be placed in actor partitions such that the actor can be scheduled correctly.

Generalized Dataflow Scheduling A different but related scheduling approach was presented by Plishker et al. in [107]. This scheduling approach is based on a dataflow MoC called Core Functional Data Flow (CFDF) [106], which describes a dataflow actor as a set of modes which have static token rates. The dynamic behavior of an actor is described by invoking one mode which gives the next valid modes; the modes and transition between these could simply be described as an FSM. SFDF can easily be used to describe MoCs such as SDF by having only one mode, CSDF by having a set of modes that are cyclically enabled, and BDF by having one mode checking the Boolean value and enabling one of the two modes corresponding to the Boolean values *True* and *False*.

The modes of the actors in CFDF have static token rates, like actions in CAL, and the scheduling of the modes is a result of a mode being invoked, resembling the action scheduling in CAL. So, it is obvious that this approach is applicable to CAL.

The idea in the scheduling approach is that, while an actor may have dynamic behavior, the modes are static and may be statically composed with modes from the other actors. These compositions, then, are static dataflow graphs which interactions can be scheduled dynamically. The result is a quasi-static scheduler, which dynamically searches for a static graph that can be invoked.

A Rule-Based Quasi-Static Scheduling Approach In many cases, a dataflow program may consist of many static actors while a few dynamic actors are needed to take care of any dynamic property. In such cases, composition of static actors that are mapped to the same processor core implies improved performance of the program. Falk et al. presented a rule-based

quasi-static scheduling approach in [55], where static actors are clustered together to form composite actors such that the quasi-static schedule of the composite actors guarantee deadlock-freedom.

The work by Falk et al. is more explicit, compared to this work, regarding how the schedules should be produced such that deadlock-freedom can be guaranteed globally. As we know, deadlock-freedom is not closed under composition for models such as SDF; for this reason the composition is performed according to a set of rules which produces a quasi-static scheduler for the static actors such that deadlocks are avoided. This is similar to how a CSDF actor can avoid deadlock, by splitting the firing in to phases, where an SDF actor would cause deadlock. The scheduler in this model, however, is more flexible and is described as an FSM.

What this approach has in common with the approach presented in this thesis, is that there is a set of rules for when actors can be composed, and then there is one or more strategies for performing the composition and scheduling. The difference is that this approach in general produces composite actors by adding dynamism to the original static actors, while the presented approach only removes dynamism. A scheduling strategy which adds dynamism for avoiding deadlocks would be useful for the approach presented in this thesis, as it would enable more options for constructing partition for actor composition.

Model Checking Dataflow Applications While model checking typically is used to verify that a design behaves correctly, using model checkers for finding a better solution to various problems has also been used to some extent. In [109] Ruys presents how the Spin model checker can be used to find optimal solutions for problems like traveling sales man, which to some extent resembles the problem of scheduling a dataflow program. Here the guarded commands includes cost functions, that is, depending on which choice is made, a different penalty is added to the total cost. Every time a better solution is found, that is, with a lower cost, Spin emits a trail with the *schedule* of the traveling sales man.

The traveling salesman problem is to some extent similar to the CAL scheduling problem, as soon as a complete model has been generated for a partition of a CAL program. An interesting idea, would be to use a similar approach as presented in [109] to define cost for the order of actions to not only search for a schedule, but to, for example, search for a schedule with better data locality.

Using model checking for analyzing and optimizing dataflow applications is not a completely new idea either. In [115] Theelen et al. present results of analyzing scenario-aware dataflow with the CADP model checker, to formally verify functional and performance properties of the dataflow model. Also some work related to scheduling of dataflow has been pre-

sented; scheduling of SDF with the goal of minimal buffer sizes has been presented by Geilen et al. [59], Liu et al. [87], and Guan et al. [62], and with extensions to CSDF, MD-SDF, and BDF, by Gu et al. [61].

Chapter 6

Discussion on Efficiency and Implementation

Actor composition and scheduling is not the goal, in itself, but are means to transform the program into a more efficient format to better fit a target platform. To improve the performance of a program, several design choices affect the end result and depending on the target platform and what actually is meant with performance, some choices are more adequate than others. The goal with this chapter is to present some experimental results regarding actor composition and draw some conclusions from these to better understand how a dataflow program should be designed and transformed, such that efficient code can be generated.

From the scheduling perspective, the performance is affected by the complexity of the scheduling, meaning that the number of guard evaluation and the complexity of these, decide how much work is spent on scheduling. The partition sizes also play a role here as the number of partitions decides on how many partitions must be scheduled by a run-time scheduler choosing the actor partition to schedule next. At the same time, the partition sizes also affect the memory footprint of the program which affects cache behavior. This means that a longer static schedule, while requiring fewer guard evaluations per action firing, also requires more memory when running.

If composition is a method to increase the size of the actors and thereby to improve the performance of a program, one could imagine that it would be easier to directly design the program with large enough actors in the first place. And, this is true; the performance of a program that has been tailored to fit a platform is difficult to achieve by transforming another program. However, implementing a program for one specific platform is not very interesting, instead, a program should be portable and a decent and predictable performance should be achievable, on many different platforms, by applying a set of transformations and optimizations. The approach cho-

sen here for this problem is to design fine grained programs and use composition to achieve coarser grained designs when needed; the opposite, that is, splitting actors, could also be possible but is not investigated here.

Ideally, a dataflow program designed for an FPGA should be designed on the level of adders and multipliers, producing a program with a maximal number of simple parallel actors. The scheduling of each of the actors is not a problem in this case, since each of the actors have dedicated hardware and there is no software interleaving of the actors. It does not make sense to compose and schedule, and thereby sequentialize such an application. On the other hand, when an application is designed for a general purpose processor, the scheduling and interleaving of the program on the level of adders and multipliers would introduce a huge overhead and the program needs to be implemented on the level of larger functional blocks in order to be efficient. It is this kind of fitting applications to processors which is the main purpose with the work presented in this thesis, the difficulty is, however, to decide what a program should look like in order to fit a specific platform.

6.1 Design Parameters

The main goal with composition of actors is to enable more efficient scheduling. First of all, the scheduling of actors, that is, the task to find an actor with an enabled action becomes simpler when there are fewer actors to schedule. Simultaneously, the number of guards that needs to be evaluated per action fired is reduced as one guard can enable a sequence of actions that can fire. These improvements of a program are, as such, platform independent, as they simplify the action selection process and by this only reduce the number of calculations that are needed in order to run the program. The results of actor composition, however, are not platform independent; instead the schedules affect the size of the work units and thereby also the memory usage. Sometimes, these properties have an opposite effect on the program performance; a composition of two actors may reduce the scheduling overhead while causing worse cache behavior. Which of the properties have a stronger result on the performance depends on the architecture of the target processor.

Considering a single processor, the number of actors that works well on the processor depends on a few things. To begin with, the program itself gives the first indication of how many partitions it is efficient to compose the program to. For more complex programs, it is often difficult to find simple schedulers for the whole program and instead unrelated parts of the program should be placed in different partitions. This is then a trade-off between the complexity of the actor scheduler and the action schedulers.

This gives the first idea of how large partitions it makes sense to create; the second trade-off to be considered is between the cost of branches in the code compared to how well the code fits in the cache. These are properties of the target platform and a transformation that makes the code faster on one platform may produce slower code on another platform.

This becomes evident from the results from Paper 3 [50], where three different platforms are used to evaluate a video decoding application, where the program is composed, scheduled and transformed, according to eight different configurations. The application has two partitions which either are composed or not in the experiments, and each of the combinations of composed partitions are evaluated with both minimal FIFO sizes and with a FIFO size of 4k. The used platforms are quite different, with the simplest being an embedded processor with only level one cache while the most complex processor is a server processor with a total of 16 MB of cache. The interesting result is that the middle processor which has a 1 MB level two cache shows opposite results compared to the two other processors when the transformations are applied to the CAL program with different sizes of the FIFOs. The informal explanation for this is that, for the processor with small cache, the cache is too small for each of the experiments, while for the processor with large cache, the cache is sufficient for all of the experiments. The processor with medium cache, however, experiences the transition from having enough cache to not having enough when the sizes of the FIFOs are increased significantly. Another surprising result, measured on the simplest processor, is that when either of the partitions is composed the speed increases, however, when both of the partitions are composed the speed is worse then one of the previous results where only one of the partitions is composed. Again, the only explanation is the memory footprint, however, the exact reason is difficult to identify as the different compositions may allow the compiler to perform different optimizations which also further may affect the memory usage.

It is difficult to predict the exact result of a composition as it is hard to predict how the compilers, caches, branch predictors, etc. reacts to the change and how these interact. For some processor types, such as Very Long Instruction Word (VLIW) architectures, which make use of instruction level parallelism to explicitly at compile-time schedule several instructions in parallel, the cost of branches is typically high [16]. In Paper 5 [26], experiments are performed on a Transport-Triggered Architecture (TTA) processor, which is a VLIW like architecture where the parallel instructions are scheduled explicitly in instruction words, like in VLIW, but the instructions explicitly describe the data moves on the interconnection network of the processor. On this kind of an architecture, the actor composition with quasi-static scheduling produces excellent results; long static schedules which mean that there are long code segments without branch instructions,

Figure 6.1: A simple example program with static token rates.

allow the compiler to exploit the instruction level parallelism. In this experiment, the CAL program, an IEEE 802.15.4 (Zigbee) physical layer wireless transmitter, was used to produce TTA code by using several approaches and was compared to a handwritten C-code implementation. The experimental results show that, using actor composition with quasi-static scheduling and finally merging the actors into larger actors, the program performance measured in time it takes to transmit a packet, was able to compete with the hand written C-code program. The main difference between the programs was that the code size of the composed CAL program was several times larger than the hand written program. While these are promising results, the processor type used in the experiments favors large actors and the results does not apply to processors in general.

What are the conclusions from these results? The experimental results show how different design choices affect the performance of a program depending on the target architecture. The design space exploration is then about finding the trade-off between more parallelism but with more scheduling overhead and less parallelism and fewer scheduling decisions. If the sweet spot is at either extreme or in the middle depends on both the program and the target platform.

6.2 Performance

To understand why some compositions or schedulers fit better than other for a platform, it is necessary to have an understanding of what actually happens with the program when (quasi-) static scheduling approaches are applied to it. In the long run, a dataflow program is repeatedly firing sequences of actions that results in data being streamed through the program. Different scheduling approaches may produce schedules with different behavior regarding data locality or the number of branch instructions needed. This all affects how the program performs, and, of course, the impact of these properties depend on the target architecture.

To begin with, let us consider a simple round-robin scheduler which simply traverses a list of actors and runs the action scheduler of a single actor as long as the actor has eligible actions, after which it continues with

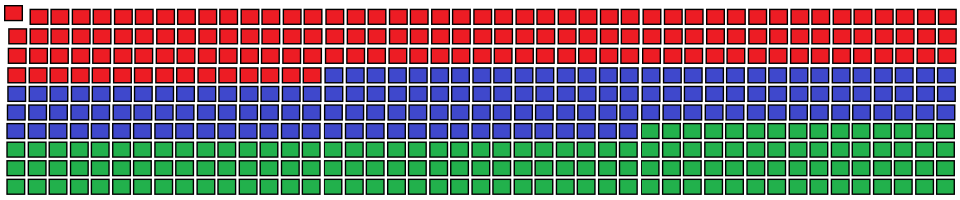

Figure 6.2: An execution trace of the example dataflow program. The trace fires each actor 150 times, making a scheduling decision between each firing. The round-robin scheduler repeatedly fires a single actor until either it the input is empty or the output is full.

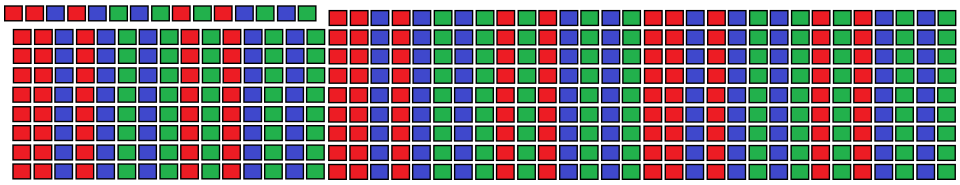

Figure 6.3: An execution trace of the example program where a static schedule of length 15 firings has been produced. As all three actors are included in the schedule, the round-robin scheduler only has one single item to schedule.

the next actor. A simple dataflow program which have three actors each with a single action is shown in Figure 6.1; an ideal action firing trace for this program, with the round-robin scheduler, if the FIFO sizes are set to 150 elements, is shown in Figure 6.2. First the red actor fires 150 times filling its output buffer with 150 elements, then the blue actor fires 150 times, consuming the tokens on the first queue and filling the second; finally the green actor consumes the 150 tokens. After this, this trace can be repeated forever. What actually happens here is that the action sequentially reads a list of input values and writes a list of output values. As the action is repeatedly fired, the code will definitely be found in the cache, also the cache lines with the data is likely to be cached as the data can be represented by a continuous memory area. A potential overhead for this setup is that the scheduling granularity is one action, that is, the action scheduler evaluates a guard for every action that is fired.

Compare this to a static schedule such as the one shown in Figure 6.3. The scheduling granularity of this schedule is to fire each actor five times, and the trace shown in the figure repeats this static schedule 30 times to produce the same amount of work as in the previous example. The schedule is simply one of the possible schedules that can be produced for this static program but the length of the schedule is arbitrarily chosen. The difference to the previous example is that the scheduler now fires 15 actions for each guard

evaluation; at the same time, within a schedule, code from three different actors are fired and both FIFOs are accessed. This puts more pressure on the cache. Which of these two schedulers performs better depends on how well the schedules fit in the cache, regarding both instructions and data, and how expensive the guard evaluation is.

Multi-core Performance While this work focuses on composition, and thereby on improving the performance of a single processor core, the goal with the work is to enable efficient solutions for platforms with more than one cores. The approach for composing and scheduling actors is directly usable of many-core systems as the only thing it does is to reduce the number of actors that must be scheduled at run-time. For the performance of one of the single cores, the same balancing between reducing scheduling overheads and improving cache behavior as above applies, however, the interaction between several cores makes the performance even more difficult to predict. Communication between actors mapped to several cores introduces performance limitations for example as a result of concurrent memory accesses [89].

The execution of a CAL program on a platform with several cores can in principle be done in two different ways. One way is to leave the scheduling of the actors to the operating system and let each actor be run on a separate thread. The problem with this approach is that the operating system does not have enough information about the dataflow program to be able to schedule it efficiently [127]. Another approach is to have one thread per core, where the thread has a number of actors that are mapped to this core to schedule. This kind of a scheduler for multi-core scheduling of CAL programs was presented by Hyviquel et al. in [127], where a distributed scheduler, based on a combination of a round-robin and a demand driven scheduling strategy, implementing the distributed scheduler by using lock-free communication channels for exchanging scheduling information, was used. With the first approach, mapping an actor to a thread, composition makes the task of the thread scheduler easier as the number of actors can be matched to the number of cores. By this, the scheduling of threads that anyway cannot fire anything is not an as big problem. When a dataflow program is run on a platform with many cores, one of the problems is to balance the load of the cores. Load-balancing can be performed by moving the execution of one actor to another core. With composition, the granularity of the load balancing is coarser which implies more coarse grained load balancing.

Considering multi-core architectures, and especially heterogeneous multi-core architectures, it is often more sufficient to use design space exploration to, at design time, come up with an appropriate mapping. With such a static mapping, the need for actor composition becomes clearer as the com-

position problem now is performed per core. As an example, each of the design space exploration approaches for dataflow to multi core architectures in [89, 108, 19, 32, 100], present a step where actors either are composed or split. The work presented in this thesis can be seen as that part of the design space exploration where the actors mapped on to the same core are optimized for better single processor performance. This still does not remove the potential problems with the communication between the processor cores; however, this issue has been addressed by, among others, Roquier et at. in [108], where the communication between the actors are handled and scheduled by adding additional vertices or actors to handle the inter-core communication.

As can be seen, the performance of a program, especially on multi-core, depends on several different details of the implementation, and, scheduling and composition, by them selves, does not guarantee efficient code. On the other hand, by using the appropriate design space explorations methods, to identify the parts of the program where composition will be adequate, different scheduling and composition approaches will make a difference in how the generated program performs.

6.3 Efficient Code Generation

The quasi-static scheduling of an actor partition enables the actors to be merged into one larger actor which has a predictable behavior internally. A composed actor includes the FIFOs connecting the actors in the partition, all the state variables of the actors, and all the actions of the actors. The interaction between these is after composition more predictable as the behavior of the actor is described by a set of static schedules. To allow efficient code generation it makes a difference how the actor is merged and transformed before code generation.

Typically the code generation, from high-level descriptions such as dataflow, targets other programming languages (e.g. C/C++) which already have a wide support for code generation for various processor architectures. This in turn, means that part of how efficient a program is depends on how well the *low level* compiler is able to optimize the code. Several trade-offs, that should be considered in the code generation is discussed by Battacharya et al. in [18], and in addition to transformations related to high-level concepts such as clustering and scheduling, also code generation aspects such as a potential negative interaction between e.g. inlining and register allocation. While the work in this thesis is concerned with scheduling and composition, it will make a difference how the merged actors are constructed. For the tool chain, this means two things: first, the transformation should be made on a close to algorithmic level as low level optimizations are provided

by the low level compiler and all possible options should be kept open for the optimizations. Second, optimizations related to the information specific for a dataflow program, should be used to minimize the needed calculations of the program as the low level compiler does not have this information.

The actor merging approach presented in Paper 5 [26] is an example of this. The quasi-static scheduler provided for an actor partition is implemented such that there is a minimal loss of information. The actions of the original actors are made into procedures, the schedules of the composed actor becomes actions which sequentially fire the action procedures, and the FSM and guard of the compound scheduler are directly used. This kind of a structure enables the code generator to inline these procedures (action) if it is useful, or to iterate sequences of repeated actions either in a loop or as a sequence of procedure calls. The point is that it is up to the low level compiler to decide what is the most adequate code for the target platform.

The actor composition enables the actor merger to remove all internal buffers of a composition as the quasi-static scheduler of such a composition can be used to predict the read and write behavior of the buffers. As a result, FIFO channels can be replaced with simple data structures such as arrays.

What should be handled by the dataflow code generator, however, is anything related to removing scheduling decisions, removing redundant scheduling information such as FIFOs or state variables, and anything related to sequentializing parts of the code. This type of transformations may not be possible for a low level compiler to perform as it cannot know about the restrictions a dataflow program has regarding data locality etc. This means that is would make sense for the tool to remove state variables and FIFOs that cannot affect the outputs of the partition that has been merged to one actor.

With the composed actors, the last step which affects the performance in the actual code generator. There exists a number of code generations for CAL, and there is a lot of development around such tools as Orcc. As these code generators are constantly improved, it is important that the, in this case external, scheduling tools have the ability to use the code generators of these.

6.4 Related Work

This chapter is mainly based on the experimental results that has been obtained from the evaluation of the scheduling approaches presented. There exists a number of publications which present scheduling approaches and each of these present, in one way or another, experimental results which could be compared to the results in this work. While it to some extent is

hard to directly compare measurements such as speedup, some experimental results should be compared to show, on a more general level, that the presented results and conclusions hold.

Some other Experiments To get more broad results regarding the potentials of efficient scheduling and actor composition, some of the approaches presented earlier should be compared regarding the performance improvement reported.

Wipliez et al. presented actor classification in [124, 123], where the scheduling is performed based on the classes of MoCs that the actors are determined to belong to. This kind of an approach can provide very efficient scheduling when the actors belong to static MoCs, which also is shown by the results is Paper 3 [50] where this classification approach was used to schedule one of the partitions. When the actors of a partition belong to statically schedulable classes, it is difficult to improve on the speedup of this type of approaches. With the actor classification, the speedup of the MPEG-4 decoder that was used as an example in [123] was about 20%. However, the results in paper Paper 3 [50] shows, for a different platform, that the speedup of an MPEG-4 decoder is up to 53% using the actor composition based on the classification presented in [124].

Boutellier et al. presented an scheduling approach based on dynamic code analysis in [27]. The dynamic code analysis resembles the work presented in this thesis, with the exception of how the input sequences of a program are generated and how the actual schedules are searched for. Else, the type of composition is quite similar as it is based on the traces of how actions of several actors are fired. The results presented in [27] show a speedup of up to 58% for an MPEG-4 decoder running on a Intel Core 2 Duo E8500 processor. The produced schedules, with the dynamic code analysis approach, result in schedules that resembles the schedules produced by the model checking approach. For this reason, the resulting performance can be expected to reside in the same range, however, the partitioning is more closely specified in the dynamic code analysis approach.

Gu et al. presented using static regions for enabling static scheduling of parts of a dataflow program in [60]. This approach, while being elegant, only performs well when the application is enough static. The approach was evaluated on the IDCT network of a CAL implementation of an MPEG-4 decoder and showed a speedup of about 10% regarding the frame rate. Again, it is difficult to compare the results, however, it shows on improvement when the dataflow network is appropriately restructured. What is interesting in this approach, although it was not the case in this example, is that the restructuring not necessarily is a pure composition but also may involve splitting actors, or ending up with a completely new set of actors.

Plishker et al. presented in [107] a *generalized scheduling approach*, which

is based on the CFDF model of computation. The experimental results presented show a reduction of execution time of up to almost 85% for some signal processing applications, compared to a simple round-robin scheduler. However, for one of the test programs, the run-time increased; this result corresponds to the findings presented in this chapter regarding the trade-off between different overheads.

Falk et al. presented a rule-based qusi-static scheduling approach in [55], where sets of static actors were clustered together and were quasi-statically scheduled such that deadlocks were avoided when the composite actors interacted with the surrounding including dynamic actor. Falk et al. presents some experimental results with the proposed clustering algorithms where, among other results, the performance of a Motion JPEG decoder was improved by 40%, when the (fine grained) IDCT network was clustered with the proposed algorithm. Another experiment showed that the speedup of a coarser grained mp3 decoder was was about 6%. This is another example of that clustering, or composition, is only useful up to a specific point where the program reaches a sweet spot, which again depends on the platform.

Each of these works show on an improvement of the speed of an dataflow program in the (quite wide) range of 10-85% depending on the platform and the application that was used in the experiment. These results correspond to the experiments performed in this work, and therefore shows that actor composition and scheduling is an important part of the tools that are needed for fitting dataflow applications onto various processor architectures. The other property that will be of interest to evaluate based on relevant literature is the scalability of dataflow program onto multi-core architectures.

Scalability to Many-Core The main reason for that dataflow is interesting for software development is that is scales well to many-core architectures. Now, while it is easy to see that the explicit parallelism of a dataflow program at least in theory scales to as many cores as there are actors, a statement like this needs some evidence taking into account the potential overheads of a realistic platform.

CAL is in principle platform independent and the same implementation should run on anything between an FPGA and a single processor architecture. Eker et al. presents in [47] an overview of results where CAL is used to target both hardware and software. One of the results show that a MPEG-4 decoder scales, almost linearly, at least up to four cores without modification. Of course, the program can be made to scale even better if the appropriate transformations are performed on the actor network.

To know which transformation to perform, some knowledge regarding the program is needed. Casale-Brunet et al. presented a tool called TURNUS in [32], which performs a profiled simulation on CAL dataflow programs. The tool records causation traces and analyses properties such as compu-

tational load, optimal buffer sizes, critical path measurements, etc. The analysis provided by TURNUS is used to decide how the program is to be refactored to achieve the requirements. [32, 3]

With the appropriate design space exploration for identifying the transformations needed for a program to efficiently run on a program, dataflow programs will provide a scalable implementation of many algorithms. The scalability to many-core, of course, is still limited by the size of the program and the communication overheads between cores.

Chapter 7

Some Scheduling Case Studies

The scheduling methods should be general enough to produce a scheduler for any program, or at least reproduce the original actor if it cannot produce a better one. At the same time, the scheduler should use specific properties of the program to be able to identify more clever schedules for the program. This is a difficult trade-off; in the work by Wipliez et al. [124], for example, the actors must conform to one of a few statically schedulable MoCs. The schedules produced are very efficient but it also requires the actors to be implemented such that these can be scheduled according the one of the given MoCs. In this work, the target is something more general, which is enough to identify the schedules to be searched for. In practice, the question is how to ask the model checker the right questions regarding the state to search for. This was already described in Chapter 5, and worked for the kind of applications that were used as examples, but it is still relevant to evaluate the methods with some more applications and types of applications.

In this chapter some relevant applications [1] for the signal processing and multimedia domain are used to show how different properties of the program affect the scheduling approach. The idea is to from this draw some conclusions about the generality of the approach and what kind of problems may arise. The two first cases are basic digital filters, where the scheduling can be expected to be static, however, the implementation may cause some difficulties with the scheduling. The following case is a network protocol application, which also may be expected to have static scheduling, but the size of the packets is variable. The last cases are a couple of video decoding implementations, representing larger applications than the other

[1] The applications are available at https://github.com/orcc/orc-apps and I would like to thank all the authors for their work as non of this work would have been possible without their contributions

case studies, and are for this reason are partitioned in to a couple of regions which are individually scheduled and composed.

7.1 Case I – FIR Filter

To get started, let us consider a very simple, and for dataflow typical application, namely a Finite Impulse Response (FIR) filter. An FIR filter is a digital filter where the output signal is a function of a finite number of inputs and can be implemented with a number of simple actors such as adders, multipliers, and delays. In practice the operation of an FIR filter is about reading input tokens, keeping the token in the filter for some time, and generating output as the sum of different input tokens multiplied by some constants depending on how old the input is. As an example, a three tap filter is a filter that uses the current input and the two previous inputs to produce the output sample.

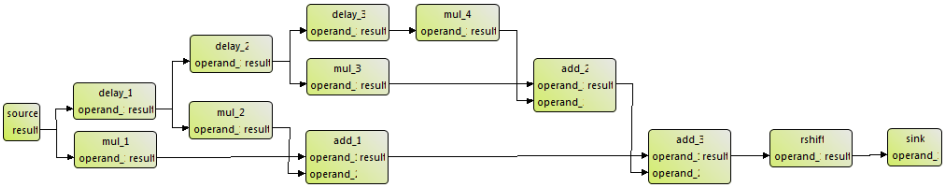

Figure 7.1: The FIR filter implementation in Orcc.

The CAL program used for this experiment is shown in Figure 7.1. The actors are rather simple: the adders, multipliers, and rshift actors, perform their respective operation in one action firing; similarly the source and sink actors produce or consume one token in each actor firing. The delay actors also consume and produce one token each firing, however, it keeps an internal buffer delaying the tokens, in this case, with one token meaning that the first input will correspond to the second output. Finally, the delay actors and the source and sink actors have initializers which must be scheduled and run before starting the actual computations; but except for this, the actors clearly have static token rates and it should be possible to find an SDF like schedule for the operation of the filter.

Two schedules can be expected to be found for this application, an initializing schedule that is run once, and a schedule that schedules one token through the filter, which can be repeated forever. In order to find appropriate schedules for the application, two properties need to hold for a schedule, first, a schedule must show some progress, that is, some actions must be fired and the actors must have been initialized, second, the FIFOs must be empty after the schedule has been completed.

The actors of this network are classified to belong to either SDF or KPN, and none of the actors have input dependent guards. In other words, the actors are not time-dependent and there are no control tokens sent between the actors. As a result, the application can be scheduled based on token rates and actor states, only. One actor is chosen as the leader for the partition, but, because there are no control tokens in this partition, the choice of actor is arbitrary and has no consequence to the composition whatsoever and only affects the naming on the schedules. In this example program none of the actors has an FSM scheduler which means that it is modeled an FSM with one state and each action is a transition starting and ending in this state.

The first schedule to search for is the one starting in the initial state and ending in a state where the program has made progress and has empty FIFOs. This schedule is found and includes the initialization of all the actors which have an initializer, but it may also contain an arbitrary number of iterations of the filter. The reason for this is that if the state space is traversed in such an order that some tokens are read into the dataflow network before a state matching the properties searched for is found, the schedule includes processing these tokens. It is not incorrect to have such a schedule; however, it means that the program cannot be initialized before a specific number of input is available. A more general schedule can be found by instructing the model checker to search for the shortest trace to such a state. The Spin model checker performs this, when it is requested, by, when finding a matching state, continuing searching for that state but restricting the search depth to be less that the shortest trace so far. By using this approach a minimal schedule to the state where only the initializer has been run is found. This state is named 's1' and in a similar fashion the minimal schedule processing one token and returning to this same state is found. The scheduler can be described as an FSM with two states and two transitions:

```
0 one_state_run one_state one_state s0 s1
1 one_state_run one_state one_state s1 s1
```

The output from the scheduling framework is an XML file describing how actors are composed and how the composed actors are to be scheduled. A simplified version of the scheduler description, which shows the relevant parts for the discussion, is shown in Figure 7.2. The scheduler includes an FSM scheduler and static schedules which here are called *superactions* and include a list of action firings and guard expressions which in this case are empty. This description of the composed actor scheduler is then used to merge the actors before the actors are code generated.

The produced scheduler consists, as expected, of an initialization phase and an SDF schedule which then can be repeated as long as the application

```
<superactor name="cluster_delay_2">

<fsm initial="s0">
<transition src="s0" dst="s1" action="s0_one_state_run"/>
<transition src="s1" dst="s1" action="s1_one_state_run"/>
</fsm>

<superaction name="s0_one_state_run">
<guard/>
<iterand action="init_act" actor="delay_3"/>
<iterand action="init_act" actor="delay_2"/>
<iterand action="init_act" actor="delay_1"/>
</superaction>

<superaction name="s1_one_state_run">
<guard/>
<iterand action="untagged_0"  actor="mul_1"/>
<iterand action="run"         actor="delay_1"/>
<iterand action="untagged_0"  actor="mul_2"/>
<iterand action="run"         actor="delay_2"/>
<iterand action="untagged_0"  actor="mul_3"/>
<iterand action="run"         actor="delay_3"/>
<iterand action="untagged_0"  actor="mul_4"/>
<iterand action="untagged_0"  actor="add_2"/>
<iterand action="untagged_0"  actor="add_1"/>
<iterand action="untagged_0"  actor="add_3"/>
<iterand action="untagged_0"  actor="rshift"/>
</superaction>

</superactor>
```

Figure 7.2: The scheduler for the FIR filter.

is run. Of course, this is a very simple program which is known to be SDF and can for this reason be scheduled with SDF techniques as long as the initializers can be handled separately. However, at the same time, this shows the strength in the generality of always describing the scheduler as an FSM; it does not matter if there are some actors that must be fired before the scheduler starts operating like an SDF scheduler.

7.2 Case II – IIR Filter

A slightly more complex application, an Infinite Impulse Response (IIR) filter, makes the scheduling more complex by requiring the dataflow network to handle delay tokens. An IIR filter, compared to an FIR filter, uses previous output samples, in additions to input samples, to produce the out-

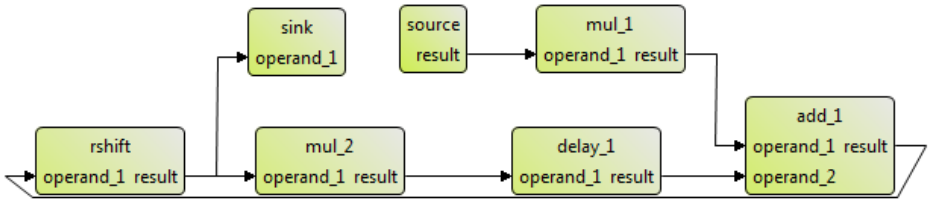

Figure 7.3: The IIR filter implementation in Orcc.

put. Using output values to produce new output means that is is possible to design a filter where an input impulse affects all future outputs. The CAL program to be used in this case study is shown in Figure 7.3; what is interesting with this program, from a scheduling point of view, is that it is expected to have static token rates but it also has a feedback loop implying that delay tokens are needed.

To schedule this program as SDF means that the requirement to have empty buffers between running the schedules is too strict as there will always be delay tokens present in the dataflow network during the normal operation of the filter. The delay tokens, in this specific program are implemented in the *delay* actor, which after running its initializer runs an action outputting a number of delay tokens before it moves on to its normal operation to simply send tokens forward. This behavior is implemented in the action scheduler of the *delay* actor:

```
schedule fsm s_init:
  s_init   ( init )  --> s_delay;
  s_delay  ( token ) --> s_delay;
  s_delay  ( run )   --> s_run;
  s_run    ( run )   --> s_run;
end
```

The scheduling of this program is quite similar to that of the FIR filter in the previous case study. The main difference is the *delay* actor which has a sequence of action firings before it starts acting in an SDF fashion, and produces the delay tokens as a result of this. The program is, therefore, expected to require two schedules, one that initializes the actors and runs the actions producing the delay tokens, and another SDF like schedule that represents the normal operation of the filter. To make the schedules make sense, the schedules are produced with the requirement that the program has been initialized, that progress has been made, and that all internal FIFOs except the output of the *delay* actor are empty. The output of the *delay* actor is a natural choice since the other actors have static token rates while this actor for every N tokens it has consumed has produced $N + D$ outputs, where D is the number of delay tokens.

```
<superactor name="cluster_add_1">

<fsm initial="s0">
<transition src="s0" dst="s1" action="s0_one_state_untagged_0"/>
<transition src="s1" dst="s2" action="s1_one_state_untagged_0"/>
<transition src="s2" dst="s2" action="s2_one_state_untagged_0"/>
</fsm>

<superaction name="s0_one_state_untagged_0">
<guard/>
<iterand action="init" repetitions="1" actor="delay_1"/>
</superaction>

<superaction name="s1_one_state_untagged_0">
<guard/>
<iterand action="token" repetitions="1" actor="delay_1"/>
</superaction>

<superaction name="s2_one_state_untagged_0">
<guard/>
<iterand action="untagged_0" actor="mul_1"/>
<iterand action="untagged_0" actor="add_1"/>
<iterand action="untagged_0" actor="rshift"/>
<iterand action="untagged_0" actor="mul_2"/>
<iterand action="run"        actor="delay_1"/>
</superaction>

</superactor>
```

Figure 7.4: The scheduler for the IIR filter.

The scheduling, however, produces three schedules. The reason for this is that the first schedule, where the requirement is progress and initialized actors, is satisfied when the *delay* actor has been initialized, the action producing the delay token is not required to satisfy this condition. The second schedule is produced in a similar fashion, and the condition is satisfied when the delay token has been produced. The third schedule, however, requires inputs in order to have progress which makes the program run one iteration of an SDF like schedule. The generated schedule is shown in Figure 7.4.

It is, to some extent, unclear how delay tokens can be identified in the general case, as it requires analysis of the token rates between the actors, which becomes more complex when the application to schedule is more complex than the simple filters that have been analyzed in these two case studies. It is, however, not required that these are exactly identified but instead it is enough to have an approximation that tells that some FIFOs most probably

may need to contain delay tokens and therefore should not be required to be empty.

7.3 Case III – Zigbee Transmitter

In this case study a CAL implementation of an IEEE 802.15.4 (ZigBee) transmitter is evaluated for how quasi-static scheduling techniques can be used to remove unnecessary guard evaluations when the individual actors are composed to form a single larger actor. The CAL program can be seen in Figure 7.5; the application, the IEEE 802.15.4 transmitter, sends packets consisting of a header and a payload of the size 5 to 127 octets, according to the standard specification.

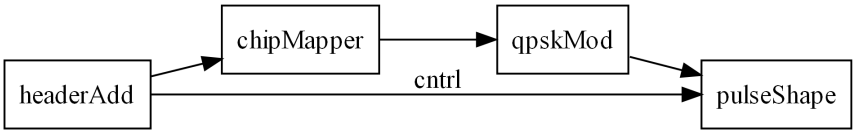

Figure 7.5: The ZigBee transmitter dataflow representation.

The resulting graph from the static analysis is shown in Figure 7.6 and some additional information is given in Table 7.1 where the actors are numbered according to Figure 7.5 from left to right. What is interesting in this graph is the dependency chain starting from an external value (left), which is used to give a value to the state variable *octet_count* which is used in guard expressions but also is used to produce a value to the port *len* which in turn is used to give a value to the *iterations* variable in the other actor.

The program, therefore, has the most complex form of control token dependency, that is, a variable which is a function of both inputs and itself, which means that it potentially may have an infinite memory of inputs. The scheduling strategies presented will not be sufficient for this application, but it is still useful to come up with a special strategy to schedule this program, in order to understand why it is problematic and how the program could be modified to avoid the problem.

With a straight forward approach, each possible input value is identified and one schedule is generated for each of the possible values, 5..127 in this case. This would be easy for the program in this study and one single guard would be enough for transmitting one packet. On the other hand, the code size would be large as the program would need to include 125 schedules consisting of hundreds of action firings. A more general approach, which we will use, is to generate the schedules between the states where these input dependent guards are used. This will result in no redundant schedules but will require slightly more guard evaluations.

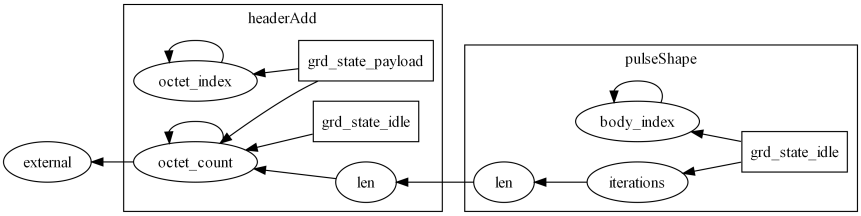

Figure 7.6: State Variable dependency graph where the arrows show from where a variable gets its value and on which variables a guards depends.

Actor	FSM	State Variables	act	choice states
actor1	4 states	count, index	7	payload
actor2	-	-	1	-
actor3	-	-	1	-
actor4	2 states	iterations, index	3	idle

Table 7.1: The scheduling information of the dataflow program showing which actors are implemented using an FSM and which state variables are used in scheduling.

The variable dependency graph in Figure 7.6 shows the actors which, does not only have token rate dependencies, but also have control values sent between the actors. When there are control tokens in the dataflow network, it is necessary to determine the possible values for these. If a control value enters from the outside, we must either from some specification find the range of allowed values or find a minimal set of guards that describe the scheduling properties related to this input. This can be done by inspecting the variable dependency graph, by verifying that the actual input values are not propagated through the network and used in a guard in another actor; if they are either the guards using these must be evaluated run-time or the variables must be used to describe the state of the scheduler.

The analyzed program consists of two actors with no input value dependency and two actors which depend on an incoming value where one of these actors pass this value to the other, this is illustrated in Figure 7.6. These control values are used to determine the number of times the loops in the FSMs will execute before continuing the execution, this corresponds to the *send_payload_octet* and *done* transitions in Figure 7.7 and to *tx_body* and *tx_tail* in Figure 7.8. The result from the analysis is that there are two FSM states in the program where dynamic behavior will appear, as it is not possible to in advance know how many times the actions in the loops will be run before the program continues.

The second step is to find a minimal set of scheduling decisions that must be taken at run-time. The starting point for this is the scheduler of

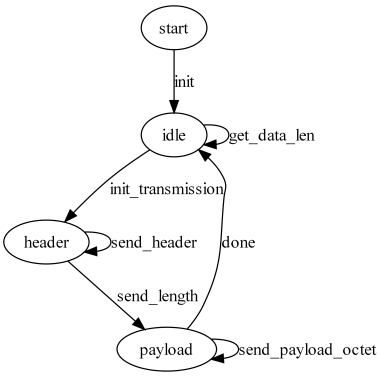

Figure 7.7: The FSM of the HeaderAdd actor.

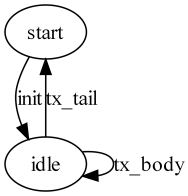

Figure 7.8: The FSM of the PulseShape actor.

actor1 as this actor obviously is generating the input for the other actors, and from this we choose the initial state plus the states where the choice between outgoing transitions depend on values not available from inside the actor. This actor scheduler is described by the FSM in Figure 7.7. From the analysis we know that the choice of transition in the states *idle* and *payload* depends on data values from outside the actor, in this case actually the size of the packet that will be transmitted, which are not known at compile-time. For the state *header*, on the other hand, the variables used in the choice are only updated within the actor and can be resolved at compile-time; as a result, this state will be unnecessary in the generated scheduler shown in Figure 7.9. Here, the transitions entering and exiting the state *header* as well as the self loop transition are represented as a single transition. The guards using the state variables in actor1 are present in the scheduler and are used to choose which schedule to fire next. The guards of actor4 are not present in the scheduler and therefore the state variables in this actor must be considered a part of the scheduler state. This means that, if we reach a state in the scheduler FSM for the second time with a different value of these variables, it is considered to be a different state or the two states must be proved to have the same scheduling properties.

Finally the last step is to generate the static schedules that link these states and form paths between these where no conditions need to be evalu-

ated. We can see one example of this from the FSM in Figure 7.7, where the transition firings needed to, from the state *idle* reach the state *payload* can be determined as the number of times the transition *send_header* will fire is known at compile-time. This is represented as *s1* in Figure 7.9; in addition these firings will, most probably, produce some tokens on the FIFOs which the other actions can process which are also represented by *s1*.

The actual scheduling is simply about finding a path between a set of states. For this we generate the model-checker for the program and are able to set the values of the four state variables and the two FSMs representing the scheduling state. We also create an input generator producing the *control input value* to the program, by generating a value in the range 5..127 according to the specification of the program. The model-checker is initialized with the starting point state and the state to be reached is partially described for the model-checker to search for the shortest path to, which when found is reported with the exact state and a trace describing the schedule between the states.

Searching for the state we only define the state from the FSM of actor1, the FIFO contents and the evaluation of guard expressions in the current state. The scheduler can, according to the previous steps, be described as the FSM in Figure 7.9. The first schedule to be found, s0, starts from the initial state of the program and ends in the state where *actor1* is in state *idle* and every FIFO is empty. The first row in Table 7.2 describes the initial state while the second describes the state after the first schedule ($s0$) has been found. To generate the next schedule, the model-checker is initialized with that new state and the next search finds the state *payload*.

The state *payload* has two outgoing transitions which depend on the input value, the objective is to found the schedule describing the self loop and the schedule that ends at the *init* state. One of these transitions is immediately enabled while the other is not, for the first we can therefore directly generated a schedule while for the other we need to first search for the state where this transition is enabled, *s3* is used for this and is for this reason not in the final scheduler, instead *s4* starts from this state and returns to the state where the next packet will be read.

With these schedules, we can see that for *s2* which starts and ends in the same state, that the variables of actor4 have different values after the schedule. In order to not have to create a new state for each possible value of these variables, we must instead prove that we can predict the behavior of actor4 from actor1.

The scheduler for the program is described as an FSM based on the FSM of actor1, while the state variables in both actors are omitted and only used to choose between the schedules *s2* and *s4*. We have the set of schedules for one of the possible inputs and in these, *s2* represent the loops in both actors and *s4* represents leaving the loop. For these schedules to be correct,

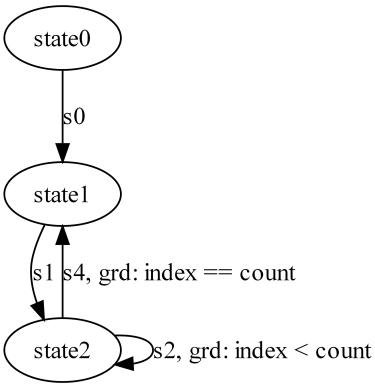

Figure 7.9: The desired scheduler, the states of the FSM corresponds to the states of the headerAdd actors states where a input dependent guard must be evaluated and the initial state.

	actor1			actor4		
id	FSM	count	index	FSM	iter	index
-	start	0	0	start	0	0
s0	idle	0	0	start	0	0
s1	payload	11	6	idle	352	192
s2	payload	11	7	idle	352	224
s3	payload	11	11	idle	352	352
s4	idle	0	11	start	352	352

Table 7.2: Program state after each step of the scheduling.

we must prove that actor1 can be used to predict when actor4 leaves the loop state.

As the input dependent guards of the first actor are used in the actual scheduling and the guards are actually evaluated, we can be sure that the behavior of this actor is correct. For the fourth actor, the guards depending on the input is omitted in the generated scheduler and the assumption made earlier was that, for a fired sequence in actor1, actor4 will always answer with the corresponding firing sequence. The two variables control when actor4 is allowed to leave the state *idle*. The only thing that must be proven, is that, the behavior in the schedules, where actor4 leaved the *idle* state only in schedule *s4*, that this also hold in the general case, for any input, and for any number of packets transmitted.

We need to define some conditions that will be useful for checking these properties. One useful property is that each message queue in the program is empty, in other words, there are no tokens produced by *actor1* that are waiting to be consumed.

$$E \equiv empty(Q_1) \wedge empty(Q_2) \wedge ... \wedge empty(Q_N) \tag{7.1}$$

The second property that we want to check is if the actors are in the state with the loop, that is, in the state where the decision whether to continue with the next iteration of the loop or to continue the program leaving the loop state. We define these states for the two actors as:

$$\phi \equiv headerAdd_{FSM} = payload \tag{7.2}$$

$$\psi \equiv PulseShape_{FSM} = idle \tag{7.3}$$

Also the condition whether to continue or not is needed, this is used to check, while still being in this state if the next state will be in the loop or after the loop, and is defined as:

$$\gamma_h \equiv grd(done) = \textbf{true} \tag{7.4}$$

$$\gamma_p \equiv grd(tx\_tail) = \textbf{true} \tag{7.5}$$

Next we must express the properties that must hold throughout the lifetime of the program. These are expressed as Linear-time Temporal Logic (LTL) statements, where we describe simple rules that must hold for each state of the program where the $\Box$ operator means each statement. The first expression says that while both actors are in the loop state and every queue is empty, then either both or neither of the actors is allowed to leave the state. The point with this check is to prove that the input dependent guards of actor4 relates to the guards of actor1.

$$\Box\,(\phi \wedge \psi \wedge E \implies (\gamma_h \wedge \gamma_p) \vee \neg(\gamma_h \vee \gamma_p)) \tag{7.6}$$

The reason why E is used is that, the dataflow program has buffers and the actors are allowed to run out of sync. When a property is checked in a state where the buffers are empty, we know that the actors are in the same iteration – in this case, transmitting the same packet. As the schedules constructed, sends one packet completely before starting to send the next one, by requiring the buffers to be empty after each schedule, we are not interested in the states where the FIFOs contain tokens.

What we know from the previous test, is that, when actor1 has produced tokens and actor 4 has consumed these, if actor1 is can run according to *s4* then so can actor4. What we do not know, is, what is allowed to happen inside a schedule, that is, when $\neg E$. This can also be described as: we have checked the correctness in the states where the schedules are chosen, but we still need to check the behavior during the execution of *s2*. The

schedule	actor1	actor2	actor3	actor4	total
s0	1	0	0	0	1
s1	8	6	12	192	218
s2	1	1	2	32	36
s4	1	0	0	1	2

Table 7.3: For each schedule the number of actions that can fire based on evaluating one single guard of the generated scheduler.

next expression simply says that when we start any schedule *s2*, *tx_tail* will not become enabled before *done* becomes enabled ($\mathcal{U}$ is the *strong until* operator).

$$\Box\,(\phi \wedge \psi \wedge E \wedge \neg\gamma_h \implies \neg\gamma_p \,\mathcal{U}\, \gamma_h) \tag{7.7}$$

We run the model-checker such that the input data value is allowed to take any of its possible values. As these properties were proven to hold, we know that the loops are synchronized and that it is possible to run every iteration of the loop with the same schedule, for the whole network.

The quasi-static scheduling of a dataflow program presented here is used to minimize the number of guard evaluations that needs to be executed during run-time. Generally speaking, each action needs to evaluate one condition including a guard expression and token availability. For this reason, a simple measure of the improvement, is the number of actions that can be fired *for free* based on one guard evaluation in the quasi-statically scheduled actor. The results are presented in Table 7.3 where we can see the number of action firings in each schedule corresponding to the transitions in Figure 7.9.

One execution of the program, that is, sending one packet, executes each of the schedules ones except *s2* which is executes 5..127 times. For each of these cases, the number of guards evaluated for sending a packet ranges from 2% to 2.7% of the original number of guard evaluations. These numbers will not give accurate information regarding speedup as the cost of evaluating guards highly depends on the platform.

While it is rather easy to determine which states need run-time scheduling and which variables result in problems regarding state space. It is not obvious which properties must be checked for a program for one scheduler to be correct in the general case. For this to be more useful, either the programmer should describe or the compiler identify, program constructs, similar to design patterns, for which conditions to check are already available.

How to Simplify Scheduling The task of scheduling this CAL program is somewhat to complicated. The actual problem is that the control token

which is sent by the *headerAdd* actor depends on a variable with complex behavior. In order to make the scheduling of this program automatic, this dependency must be removed. A simple solution is to, instead of sending a value representing the number of octets to be sent, simply send one control token for each octet, and finally one that indicates that the packet has been completed. This way the control token can be based on simple constants (true, false), and there is no control token dependency between the choice in the *pulseShape* actor and the inputs of the partition. Of course, we increase the communication of the two actors, however, in composition this FIFO can be removed as its value is not anymore used by the quasi-static schedules.

7.4 Case IV – MPEG-4 Decoder (Coarse Gained)

The next case study is an MPEG-4 [121] decoder. This program consists of three networks namely motion estimation, texture coding, and the parser. The parser is rather dynamic as it reacts to the contents of the input stream, the other two blocks, however, perform certain operations on macro blocks or 8x8 pixel blocks, and it can be expected that some static schedules should be possible to derive for these networks. The dataflow design of the program can be seen in Figure 7.10; in general, the control information is distributed on the channels named $BTYPE$ while the other channels are data queues.

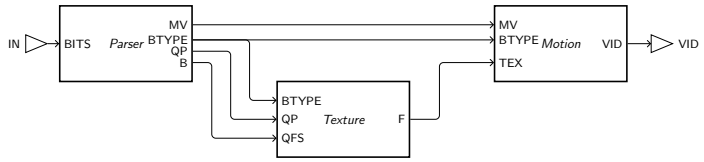

Figure 7.10: The MPEG-4 decoder.

Before going into the actual case study, it may be useful to describe the general operations of the blocks to be scheduled. Both the *Texture* and the *Motion* networks receives an 8x8 block together with the information regarding how to decode it on the $BTYPE$ input. The texture network decodes the actual pixel data and adds the potential prediction from surrounding blocks while the motion compensation potentially adds prediction from previous pictures according to the motion vectors (MV). The information regarding which predictions should be performed on the current block and whether or not the block has actual DCT coded coefficients that are to be inverse transformed by the texture network, is encoded in the values transmitted on the $BTYPE$ channel.

What we should be expecting from this program is that, for each of these networks, a schedule should correspond to some value on the $BTYPE$ channel, together with some data input on the other input ports. The two

networks, on the other hand, are not expected to be practical to be scheduled as one unit as the type of operation performed by these are different and are not dependent. That is, the fields that are checked in the *BTYPE* token are different for the two networks; this also becomes evident when the compatibility of the guards is analyzed.

Texture Coding The first network to be scheduled is the *Texture* coding network, which consists of five actors. These actors, does, according to the MPEG-4 standard, perform DC coefficient prediction, re-ordering of coefficients, AC prediction, inverse quantization, and inverse DCT. What can be assumed about this sub-network, as everything relates to decoding texture data, is that the operations performed by the actors depending on the type of block entering the sub-network, to some extent correlate. The actors can therefore be assumed to share the properties of the blocks that enter, however, this qualified guess needs to be verified more formally.

The variable dependency graph in Figure 7.11 shows that the partition has two actors which depends on the input to the partition, while one of these actors also generate control values for two other actors in the partition. To successfully schedule the partition, a set of guards that completely describes the behavior for any possible input needs to be found. The guards of the two front actors which reads the control values entering the partition checks properties of the *BTYPE* control token regarding whether the incoming block is an INTRA block, a motion block with or without residual coefficients, or perhaps the marker of a new frame. Of the two actors reading the incoming control token, one of the actors has these four alternatives while the other one has only three of these, with one of the guards corresponding to two of the guards in the first actor. The actor called *DC_Reconstruction_invpred*, is therefore considered to cover the scheduling decision of these two actors. In a similar fashion, the same actor is resolved to cover the two actors to which it generates the control values. In this

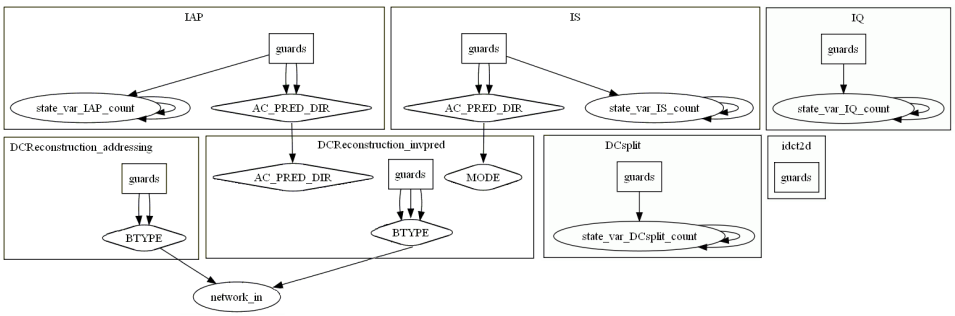

Figure 7.11: The texture network's dependency graph.

particular actor, the control values are generated from a set of constants, however, in one of the actions, one of several constants are used depending on an if-statement. This results in that each of these constants must be checked to be accepted by the same actor in order for the guard of that specific action to be strong enough.

Using the actor *DC_Reconstruction_invpred* as the starting point of the scheduling, and using the initial schedules generated according to Chapter 4, a set of four schedule to be searched for is identified as shown in the following listing.

```
0 read_read_other read read s0 s0
1 read_read_inter_ac read read s0 s0
2 read_start read read s0 s0
3 read_read_intra read read s0 s0
```

An initial state named *s0* is identified, and each of the schedules can be generated such that it reaches this state. This means that the resulting schedule only needs to have four schedules, and is therefore an FSM with one state and four transitions. Each of these transitions corresponds to evaluating one guard expression and firing a number of actions; for the one named *other* 6 actions, for *inter_ac* 330 actions, for *intra* 331 actions, and for *start* 7 actions. These numbers does not tell anything about the speedup of the program, but shows how much unnecessary scheduling decisions can be removed if this partitioning is chosen.

Motion Compensation The motion compensation sub network has a similar structure, and is controlled by the same control signal, namely *BTYPE*. While this is the same control value as used in the texture network, the properties of the control value checked is somewhat different in this network, as we now are interested in properties related to inter prediction while we before were interested in properties related to intra prediction.

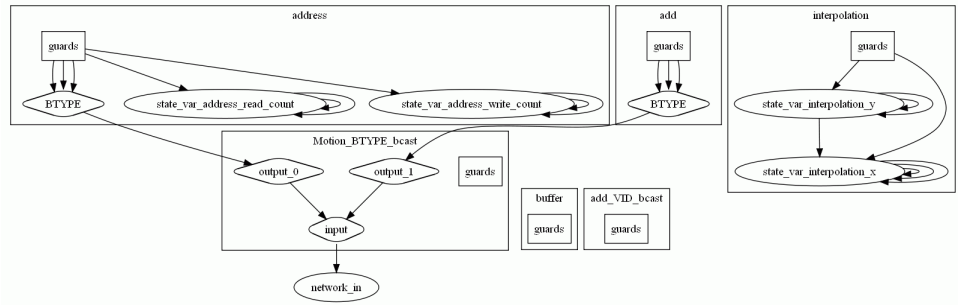

Figure 7.12: The motion compensation network's dependency graph.

For this reason, these networks are chosen to be scheduled separately, although it would not be impossible to schedule them together either, only the scheduler would become more complex.

More complex guards than what is available in any single actor will already be needed to schedule this partition. The reason for this is that the two actors sharing the input control token does not have compatible guards. In Figure 7.12, one can see that the incoming control token is used by the actors *address* and *add*. While both of these actors have four guards checking this token, both of them have two separate guards that may be enabled simultaneously with two guards of the other actor. Consequently, in order to find strong enough guards for the partition, a set of new guards, corresponding to the different possible combinations of guards of these two actors, must be generated.

Considering two sets of guards G_{add} and $G_{address}$, the needed guards corresponds to the Cartesian product of these sets, that is,

$$G_{partition} = G_{add} \times G_{address}$$

where the contents of these sets are

$$G_{add} = \{nvop, texture, motion, both\}$$

and

$$G_{address} = \{nvop, no\_motion, motion, neither\}$$

The guards needed to schedule the partition is then defined as all these combinations except the ones that cannot be accepted simultaneously.

$$G_{scheduler} = G_{partition} \setminus G_{impossible}$$

Where the impossible combinations are those, for example, requiring a block to be both intra and inter block simultaneously.

$$\begin{aligned} G_{scheduler} = \{&(nvop, nvop), (texture, no\_motion), \\ &(motion, motion), (motion, neither), \\ &(both, motion), (both, neither)\} \end{aligned}$$

The obvious problem is that no tool could ever know which of these combinations are impossible, unless the programmer adds this information. The set of guards of an actor describes the conditions that are sufficient to fire actions, but this information is based on assumption the programmer has made and is not visible in the guards. Some hints may be possible to derive from how a control token is generated but this is not always possible either. For this reason, the only way to succeed with the scheduling of this partition, such that the scheduler does not include impossible schedules, is

to add this information manually. This can be done by adding an actor, with guards corresponding to these composed guards, but does not perform any calculations; this actor can then be used to generate the scheduler of the partition.

Once the guards are in place, an initial scheduler can be generated, and the actual scheduling can take place. The six guard expressions in $G_{scheduler}$, are used to generate a simple initial scheduler with one state and six transitions, corresponding to the guards. The network is then scheduled with the following objectives: 1) consume all tokens on control inputs, and 2) all buffers inside the partition must be empty in the end state. Each of the schedules are found by the model checker, and the end states correspond to the initial state, resulting in a simple scheduler FSM with one state and seven transitions. The produced schedules fires sequences of actions: nvop 6 actions, texture 133 actions, and each of the other schedules working on inter blocks about 380 action each.

7.5 Case V – MPEG-4 Decoder (Fine Grained)

As a comparison, let us look at another finer grained implementation of MPEG-4. On a high level, this implementation has a similar structure as the previous one, and uses the same *BTYPE* control value to distribute the information regarding the type of block that is being decoded. The actors, however, describe much smaller computational nodes, as an example, in the previous decoder IDCT was implemented as one actor, while in this implementation, IDCT is a network consisting of 12 actors. While we can expect more gain from performing actor composition of this implementation on a platform with few cores, we should also expect the control dependencies to be more complex. This implementation has also been designed with hardware, or an FPGA, in mind and for this reason it also includes an actor that simulates a DDR memory.

The top-level design of this application can be seen in Figure 7.13 and be compared with the one in Figure 7.10; the difference being the DDR model and the separation of texture coding and IDCT in the more fine grained one.

ACDC Prediction Let us start with the sub network that is the most similar to the corresponding one in the previous case study, namely ACDC (or intra) prediction. The control token dependency graph of this sub network can be seen in Figure 7.14, again, it is the control value called *BTYPE* which is the control input to the partition. This value enters two actors and one of the actors resends this control value to three other actors. In order to decide if either of the actors first receiving the control token to schedule

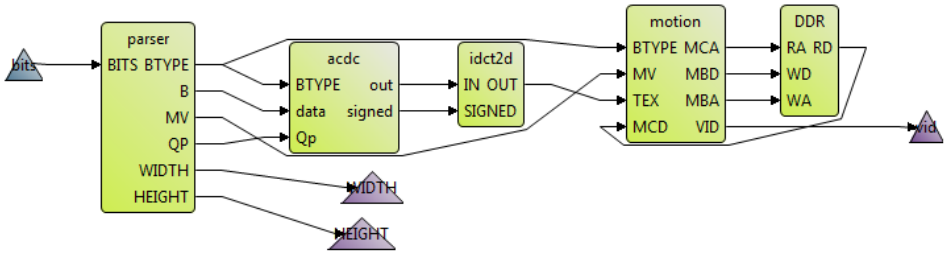

Figure 7.13: The fine grained MPEG-4 decoder as shown in Orcc.

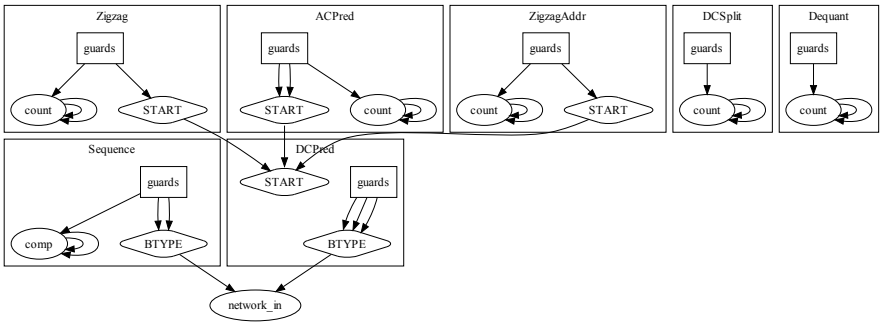

Figure 7.14: The dependency graph of the acdc network.

the whole partition, we need to decide on the guard strengths.

We need to make sure that there is a functional dependency between the guards. The actor named DCPred has four guards while the actor named *Sequence* has three guards. As both of these actors receive the control token, the functional dependency is not enough but it is required to be a surjective function. The only solution is therefore that the firing rules of *DCPred* represent the domain and those of *Sequence* the co-domain; in other words, the guards of *DCPred* could be used to schedule the partition. To be sure that this can be done, we need to compare the guards of the actors according to the methods presented in Chapter 5.

Unfortunately we cannot prove that the guards of *DCPred* are strong enough. The reason lies in the lowest priority guards which simply fire when none of the other can fire in combination with that we do not define the exact tokens that can enter in this port. For this reason it is possible to give *BTYPE* a value, which would mean that the block that entered is both *intra* and *inter* simultaneously, which in reality never could happen. To solve this, we either need to analyze the actor that produce this token or make the guards of the actors stricter. For this case study, however, we will

simply conclude that *we know* that the guards are compatible (as we know the MPEG standard).

The other control token connection, from *DCPred* to the other three actors using the *START* signal, also needs some simplification. As seen in Figure 7.14 the control value depends on constants. This is not completely true, instead, one of the actors produce one of three possible control values, which are chosen in an if-statement. The full dependency graph would, of course, include the dependency from the output port (START) to the variables that can be reached through the variables used in the condition of the if-statement. These are, however, made complex by the expression depending on a data structure which depends on several inputs. Instead, we can chose to prove that each of the constants correspond to a specific firing rule in each of the actors that use it. After finding that this is true and that the partition does not include time-dependent actors, the scheduling may begin.

As we learned from the previous steps, the scheduling should be based on the *DCPred* actor. This actor has a quite simple scheduler which can be simplified to one state where one of four schedules are fired depending on the control token and then returning to wait for the next input. A simple strategy is used, we simply require the end state of each schedule correspond to the *DCPred* actor being at its initial state *read*, each internal FIFO to be empty, and that at least one actor has made progress (fired an action). These correspond to the four first transitions in the following.

```
[0] inter : read -> read : (s0 -> s1)
[1] intra : read -> read : (s0 -> s1)
[2] other : read -> read : (s0 -> s0)
[3] start : read -> read : (s0 -> s0)
-------------------------------------
[4] inter : read -> read : (s1 -> s1)
[5] intra : read -> read : (s1 -> s1)
[6] other : read -> read : (s1 -> s0)
[7] start : read -> read : (s1 -> s0)
```

As can be seen, these four schedules end up in two different states, here called *s0* and *s1*, which means that the corresponding schedules also need to be produced for that state. What happens in practice, while it is not important for the scheduling, is that the *zigzag* actor is implemented to, in normal operation, consume and produce one token each firing. As the *zigzag* is a reordering operation, a single value is not enough to produce output, and for this reason, the actor keeps an internal buffer of one block which is filled at the first block of a frame and drained at the last block of a frame. As a result, this actor can be in one of two states after processing a block, and therefore, also the scheduler of the partition.

Guard	Nr. actions (State0)	Nr. actions (State1)
start	9	268
inter_ac	135	330
other	8	267
intra	136	331

Table 7.4: The length of the schedules generated for the acdc network.

As usual, there are more tricks to it that this. The first attempt to construct a scheduler resulted in considerably more states that two. Again, there is a practical reason for this, in this case, an action without input or output and which is used to complete the cycle of an actor's execution sequence. Depending on how the model checker interleaved the action firing, this action either fired or not before a state which corresponded to the description above was found in the state space. To solve this, the search was restricted to only search for states where the FSM states of each of the actors corresponded to one of the already identified states. While *s0* was the only known state this restriction results in not finding any matching states, in which cases the restriction is not used. Then again, once *s1* is found, no new states are needed and the restriction enforces a search where no unnecessary new states are introduced.

The resulting composed scheduler is an FSM with two states and eight transitions corresponding to schedules of different lengths as shown in Table 7.4. Each of these sequences are chosen based on the guards of *DCPred*, and as can be seen, results in removing a considerable number of unnecessary guard evaluations and thereby scheduling overhead.

IDCT The IDCT network of this actor is both complex and simple to schedule. If we know that it should consume 64 tokens and then produce 64 tokens, and we simply expect it to turn out right, then the scheduling is really simple. The control token dependency graph is shown in Figure 7.16, and it can be seen that there are no control dependencies to input ports of the partition; in other words, the scheduling does not need to pay any attention to the control tokens as long as the related state variables are used to describe the state of the partition.

The potential problem with this network is that five of the twelve actors have time-dependent firing rules, including the actor *rowsort*, which reads the input ports representing the DCT coefficients; see the network in Figure 7.15. If every connection which is time-dependent is removed, there is no connection between the two inputs. On the other hand, the actor named *clip*, which is the only actor producing output from the partition has static

token rates on its input ports. According to the rules, defined up earlier, this network cannot be scheduled, however, the *clip* actor ensures that the streams are synchronized. Setting the *clip* actor as the leader actor and viewing the other actors as a black box on one of the inputs to this actor, we can try to find a schedule for the partition. At the same time, we can also conclude that rules for whether or not the streams are independent could be made more general, however, it is not obvious how this can be done. In this case we can do it because we know that the only interesting data size on the input is 64 tokens, this again, we know because we know how MPEG works, not because on the actors.

What can be seen from this dataflow network is that, while most of the actors in the network are dynamic and time-dependent, and are classified as DPN by the approach in [124], the network as a whole, when it is provided one full block of input, can be composed into a static actor. This is to some extent related to the problem which is addressed by the MD-SDF model. If there is a way to describe that this is the interesting data size, a schedule can be derived for that size. Due to the time-dependent actors and with the analysis performed, we cannot be sure that there is no other behavior, that would be triggered with another data rate or lack of data that the static scheduling disables. In this case we just know that 64 is the interesting rate, and the partition is not inside any feedback loop which could result in deadlock if the rate is not flexible enough.

Now, to actually schedule this program, the *clip* actor, which is a static actor reading one token from *SIGNED*, 64 tokens from *IN*, and writes 64 tokens to *OUT*. An accepted schedule should then run the partition such that the token rates of this actor are satisfied and the partition is back to its initial state. We can run the model checking with an arbitrary number of inputs, of course, at least 64 data tokens, and the model checker finds a schedule matching the requirements. The produced schedule is a sequence of 756 action firings which can fire based on a guard which only checks the availability of input and the space on the output channel.

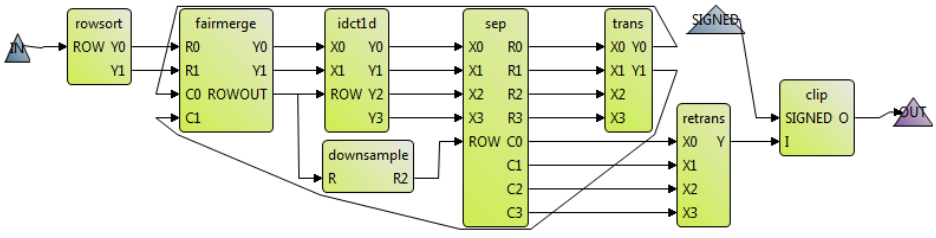

Figure 7.15: The IDCT network implementation in Orcc. It consists of seven actors and the *idct1d* subnetwork.

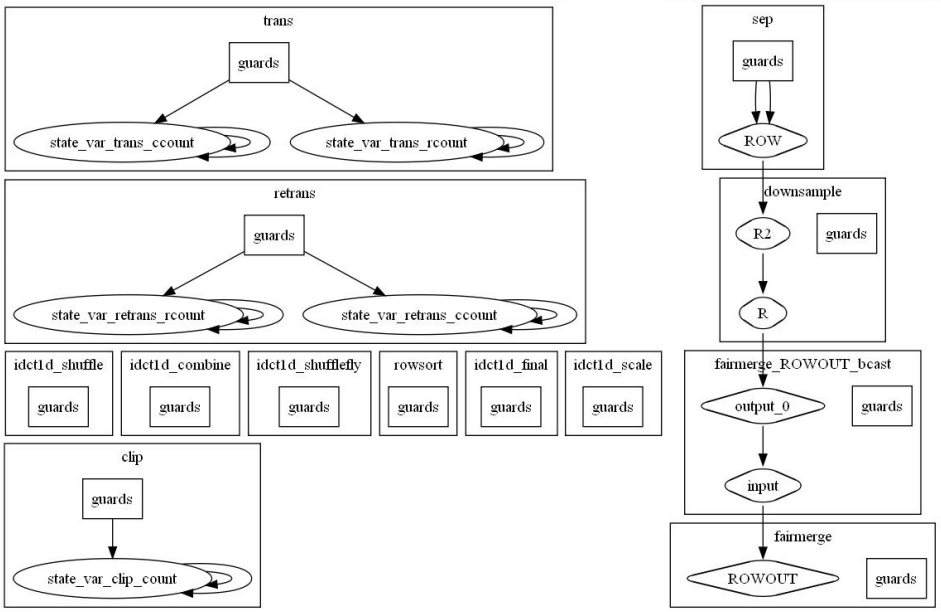

Figure 7.16: The dependency graph of the IDCT network.

Motion Compensation The previous subnetworks were problematic when it came to prove that the scheduling model is correct, the motion compensation, however, is even worse. The dependency graph shown in Figure 7.19 indicates that there is no easy way to schedule this partition. The perhaps biggest problem is the two actors which have several variables which keep a history of inputs (see the two large boxes in the figure). Either of these would be well suited as the leader actor of the partition, but, unfortunately we cannot show that either of these can be used to predict the other.

The problem we encounter with these variables is somewhat similar to the scheduling problem in the Zigbee transmitter, only more complex as there are more variables. The irony is that most of the scheduling decisions in these two actors are dependent, and, several of the variables have the same value, with the variation that they are asynchronously updated. To make things even worse, each of the actors checking the *BTYPE* control token have slightly different conditions and none of the actors' firing rules cover the others'.

Now, if these problems were solved, there is one more thing to consider. This network has a feedback loop through the *DDR* actor. If a schedule for the network would be produced without the *DDR* actor, the schedule would most likely deadlock the application as it simply would assume that the inputs that must be produced by the partition itself are available.

The alternative to composing the whole network into one actor is to

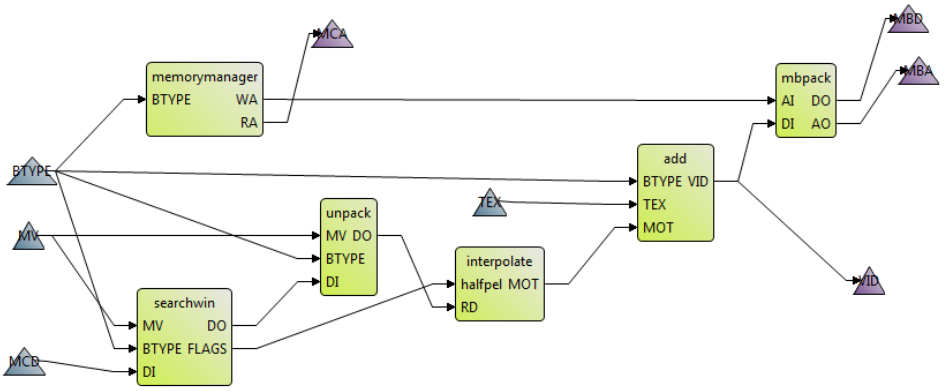

Figure 7.17: The motion compensation implementation in Orcc.

partition it in to smaller partitions that can be composed. It is obvious from the dependency graph (at least if it enlarged) that the actors *interpolate*, *add*, and *mbpack* can be composed. Of these, only the actor named *add* depends on a control token and this dependency is trivial. Compared to the two other actors, which work on blocks, the actor named *mbpack* works on whole macro blocks (six blocks). In practice this means that the *add* actor needs to consume six control tokens before the *mbpack* consumes its macro block. This means that a scheduler with six states will be needed for this composition. Another possibility is to leave out the *mbpack* actor from this composition and simply schedule that actor statically; the resulting schedule is a sequence of 385 action firings. The composite actor created from *add* and *interpolate* then requires one state and schedules including 149, 66, 149, and 3 action firings.

These composite actors are not very impressive. To add one of the two more complex actors to the composition does not make sense either, as these have such complex scheduling that the scheduler will become unnecessarily large. For instance, the actor *unpack* depends on, and has different schedules depending on where in the frame the motion vectors point. The actor named *memorymanager* is also problematic to be added to the same composition as *mbpack* as one of the outputs of *memorymanager* is connected through *DDR* and the two complex actors, to one of the inputs of *mbpack*.

Combining these three actors brakes the guarantee of deadlock freedom. The problem is that there is a data path from the partition, through actors outside the partition, and finally back to the partition. For this to not produce deadlock, the actors *memorymanager* and *mbpack* must be scheduled separately, allowing the two parts of the partition to be asynchronous. Such a scheduler would have two phases, the first runs *memorymanager* and *DDR* to produce the macro block needed for motion compensation, the other

```
<fsm initial="s0">
<transition action="s0_cmd_newVop" dst="s0" src="s0" />
<transition action="s0_cmd_y0" dst="s1" src="s0" />
<transition action="s0_cmd_other" dst="s0" src="s0" />
<transition action="s0_mbpack_data_inp" dst="s0" src="s0" />
<transition action="s1_mbpack_data_inp" dst="s1" src="s1" />
<transition action="s1_readAbove_read_none" dst="s0" src="s1" />
<transition action="s1_readAbove_read_above" dst="s0" src="s1" />
</fsm>
```

Figure 7.18: The scheduler for part of the motion network.

phase takes the block produced by motion compensation and writes back to *DDR*. This, while it works, brakes the rules of how to construct partitions, the *DDR* would actually belong to two partitions which are scheduled independently.

The resulting scheduler is shown in Figure 7.18, which is an FSM corresponding to the FSM of the *memorymanager* actor with the schedules corresponding to *mbpack* available in every state. The produced schedules are a few short ones, deciding on what operations are required for the current block. These are just a few action firings long. The schedules interacting with the *DDR* actor again are rather long: *read_above* is 297 action firings and *mbpack* is 484 action firings. With this kind of a scheduling, where there is a feedback loop, there might be a restriction of how many times the schedules of a composite actor can fire before the control is given to another actor. This means that increasing the sizes of the buffers may not enable the actor to fire more times, this in turn makes it more difficult to tune the performance of such composite actors.

What can be concluded regarding this network is that there is not much that can be done, on the other hand, with some tricks and knowledge about the actors, we can construct some composite schedulers. The tricks, however, should be translated in to scheduling strategies and the knowledge should be translated in to some specifications, for them to be useful more generally.

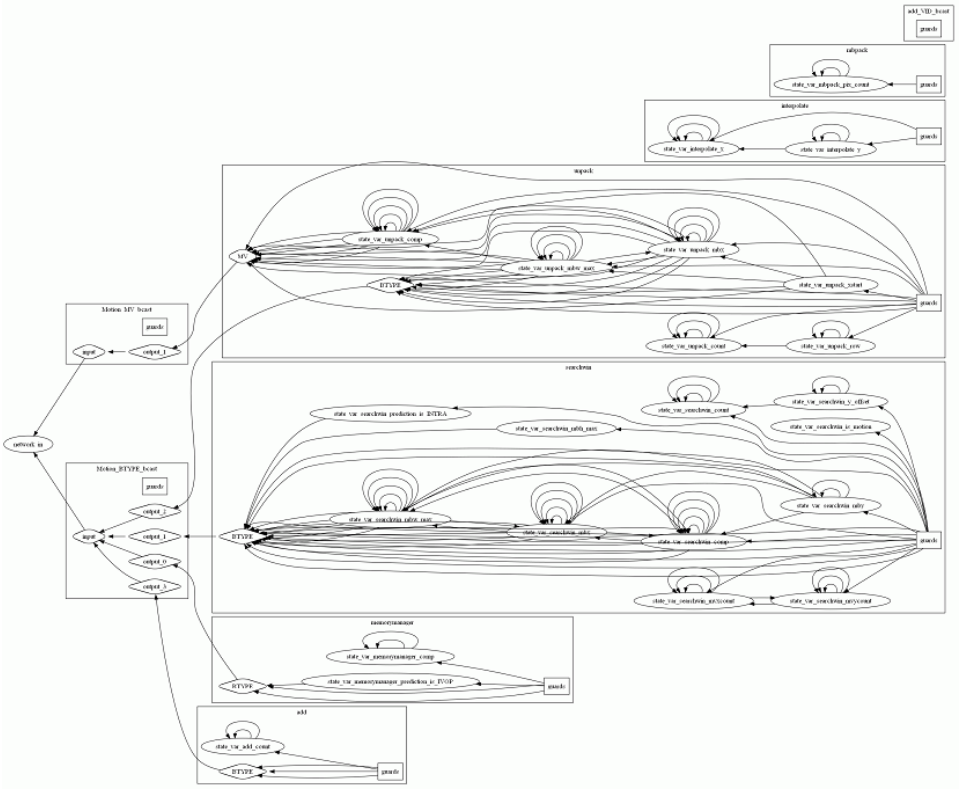

Figure 7.19: The dependency graph of the motion compensation network.

Chapter 8

Vision of the Future

The work done related to this thesis, has, to some extent shown what is possible to do and what is worth to do, and of course also what is practical to do when searching for a quasi-static scheduler for a dataflow application. Some of the experiments show how the efficiency of a program relates to the scheduling of the program, while some show what is required in order to construct a correct scheduling model of a program partition. For all these ideas, transformations, model checking, and analysis, to be practical, the tool support should be improved considerably. The purpose of this chapter is to discuss what improvements, beyond what has been presented in this thesis, would be required in order to allow any developer to use the approaches presented when developing a dataflow program with CAL.

It is my contention that the developer should describe his or her intention for the tools to be able to do something very useful. The benefit from this is twofold, fist, a well defined intention describes what should be done and gives the schedules that should be searched for, and second, by explicitly describing the intention, the tools can check if the program or part of the program does what it is supposed to do.

8.1 Identified Problems

While the purpose with this research is to show how the scheduling and composition of CAL actors can be performed using the approach presented, it is almost equally important to identify the situations where the approach was not sufficient. There are then two alternative solutions, either the methods must be improved, or the application description must be improved to include more information. Improving the description of a program does not necessarily mean that the programming language is changed, but may also simply be guidelines for how an application is described in the dataflow language.

Too Loose Specifications Many of the problems related to scheduling come from situations where the program is not specified exactly enough for the analysis to be able to draw the needed conclusions. In many cases, for a human, it would seem obvious that a guard checking if a video frame is an I-frame, that another guard checking if it is a P-frame, must be mutually exclusive, however, an analysis tool does not have the option to check the MPEG standard and can therefore not know that there are no I-frames that are also P-frames (see case study - Case V). This becomes a real problem when comparing the guards of several actors, as the intention of the programmer is not anymore available.

The source of this problem is that it seems unnatural for a programmer to overstate conditions, as an example, it would seem strange to write:

```
if (x > 0) then ...
else if (x <= 0) then ...
```

where the second condition obviously is redundant, as it can be shown that the two conditions are mutually exclusive. For a programmer, the example with the type of video frames would seem to be the same situation, however, in the case study the two properties corresponded to two different bits which either are set of not, and in that case, it cannot be known that both bits cannot be set simultaneously.

Mixed Data and Control Another related problem appears when a programmer does not clearly separate control from data. In CAL, everything is tokens, and there is no separation of control values and data values, except for the action scheduler which is strictly control. Now, consider an actor which has two actions, each producing a positive integer value. A second actor receives this value and if it was produced by action a in the first actor, it should be consumed by action a in the second actor, and if it was produced by action b it should be consumed by action b. The question is how to achieve this behavior. A clever way is to encode it in the token, e.g. action a sends the value as it is while b negates it, then the guards of the second actor can check if the input token is positive or negative. This is an example of mixing control and data, and when the control token is analyzed it is unnecessarily complex. Another alternative is to add a separate FIFO where the two actions send different constant, while the actions in the second actor has corresponding guards. This makes the analysis trivial and the FIFO can be removed in case the actors are merged.

What this is all about is, keeping the intention visible by describing control tokens with as simple constructs as possible. This improves readability for both the developer and the tools.

Distributed Complexity The presented scheduling approach performs composition of actor partitions where a few scheduling choices are identified which are enough to schedule the other parts of the partition. The basic approach is to choose one of the actors and perform the scheduling according to a simplified version of this actors action scheduler. This approach works well when there are a few complex actor surrounded by simpler actors. Compare this to the work by Falk et al. [55] where where dynamic dataflow programs, including dynamic actors, where transformed by composing the static actors into quasi-static actors, or the work by Siyoum et al. [112], where the dynamic behavior is placed in a special type of actors, as will be described later in this chapter. These approaches prefer designs where the dynamism is concentrated into some of the actors while other actors are kept as simple as possible.

An example of this could be seen in Chapter 7 – Case V, where the motion compensation network had several actors with almost identical but complex firing rules. Each of these actors, as such, could have been used to schedule a set of static actors, however, in this case it was impossible with the methods presented to construct a composed scheduler. If instead the complex firing rules had been put in fewer actors, and instead would have communicated the results of these rules by sending simple commands based on the actions fired in this actor, the scheduling would have been trivial. Of course, this kind of an implementation would reflect the original applications less as the implementation itself, to some extent, could be seen as including a scheduler.

Generality of Model Checker It is not always wanted that the token rates of all of the input ports to a partition are decided before the model checking. Instead, it is more appropriate that only the input tokens that define the schedule that is searched for are described while other inputs freely can be consumed as needed and as a consequence of consuming the defined inputs. As an example, in the composition of two actors, we might want to search for a schedule where one of the actors consume a specific number of tokens (e.g. 64 DCT coefficients), while the number of tokens the other actor consumes is not interesting in that phase but will be derived from the produced schedule. To initialize a model checker with a virtually infinite number of tokens on some of the input queues, however, will seriously affect the size of the state space and consequently the time it takes to find a schedule.

The problem we encounter here is the generality of a model checker such as Spin. Spin does not know anything about dataflow or signal processing, and it should not. However, the schedule search could be made much faster by taking into account what is known about the dataflow program or how it is expected to behave. This is to some extent related to what was presented

by [109], where Spin was made to search for a better solution according to a Branch and Bound scheme. Alternatively, a more lightweight model checker, which is an expert in dataflow and possibly implements some kind of a greedy algorithm to run the program under analysis towards the wanted state, could be implemented as part of, and integrated in, the development tools.

The difference between model checking and the scheduling approach that uses a model checker, presented here, is that conventional model checking must guarantee that an error state is not reachable while, with scheduling, we are interested in finding such an state but are not interested in other states. We also know that running a simulation will make the program end up in a state, close to that state, and often the difference between the requested state and the state a simulation would produce may depend on leaving out one action firing. It therefore seems like exhaustive state space analysis in many cases could be avoided which in turn would result in negligible development tool overhead.

8.2 Preciser Specification

To achieve more efficient scheduling, and especially more efficient scheduling analysis, the intended behavior of a program needs to be more strongly defined. There are in principle three alternatives for how this can be done: 1) by adding *descriptive* constructs to the language, 2) by restricting the MoC and using a subset of the available features, 3) by using the available constructs of the language to highlight the intention, or 4) by decorating the program with information that is not part of the actual programming language and only adds information but not functionality.

Updating a language is in general out of the question; perhaps some constructs that have been proven useful may be included in future versions of a language, but before reaching that point, the other alternatives are more useful. Restricting the MoC is an alternative that cannot be ignored, just look at all the different MoCs presented in Chapter 2, but then again, the goal with this work is to allow maximal expressive power of the language but still keep some of the benefits from inherently statical models. This leaves us with using CAL itself to produce more analyzable models, or to allow separate specifications to be added to the model.

As mentioned above, the programmer should describe control dependencies as clearly as possible in the code in order to allow efficient analysis. However, what we are interested in here is adding information without modifying the original implementation; anything added should be possible to be removed without changing the behavior of the program.

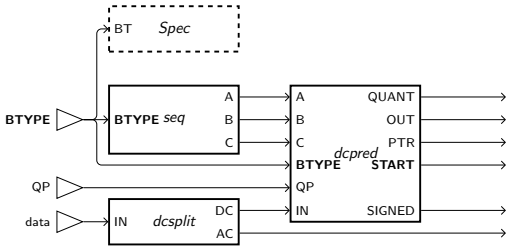

Figure 8.1: Adding an actor to strengthen the guards of the actors scheduled based on the input BTYPE.

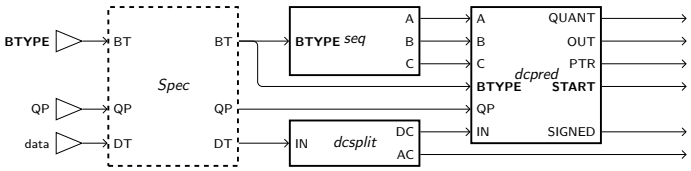

Figure 8.2: Adding an actor to define the scheduling of the rest of the partition.

Adding Scheduler Actors Let us play with the idea that we add actors that does nothing else than introduce stronger guards into a partition. An appropriate example where stronger guards could be useful is the acdc network in Case Study V, where there are two actors which guards are functionally dependent but this cannot be proved as it is possible to come up with input tokes that never will occur in practice (e.g. a frame is never both I and P), but that will cause the unwanted behavior. In this example, the network has one *control* input which is connected to two actors, namely *DCPred* and *Sequence*. These actors receives identical copies of the same control value and it is known that the four firing rules of *DCPred* can be used to decide which firing rule will be enabled in *Sequence*, however, with the given guards this cannot be proved.

An additional actor, which gives stronger guards regarding these control tokens, can be added either in parallel or in sequence with the other actors that use this control value, as shown in Figures 8.1 and 8.2 respectively. Neither of these structures add any new behavior to the dataflow program; instead they introduce a sequence of firing rules that should be strong enough to define the scheduling of the other actors. To do this the actors might include an FSM and a set of actions defining strong guards. The idea is then that, when a the scheduling model is generated, these actors can be chosen as the actors that the scheduling is based on as the firing rules have been constructed such that a functional dependency can be proved.

The first alternative, the one presented in Figure 8.1, adds the specifica-

tion actors as a *listener*, that is, it is tapped to the channel carrying control information and consumes these values. Now the question is, if this actor should be treated as any actor or if the tools should assume that this is a specific actor that specifies how the other actors work. This potentially becomes a problem if it is desired that the actor adds more information regarding, for example, the sequence in which inputs may arrive. While this type of specification actor adds a minimal modification to the dataflow network, the sequential alternative will have control over how information flows in to the partition.

The second alternative, shown in Figure 8.2, the actor connected in sequence actually resends every token that is received at the inputs of the partition. Except from defining stronger firing rules in form of guards, this version also makes it possible to add more specific firing sequences as it fully can control how the tokens are streamed in to the partition. With this construct, it is easy to specify the exact scenarios that the partition is supposed to perform. Furthermore, in this example the control token described the type of one 64-coefficient block in an MPEG-4 decoder, and it might be desired to describe that if a block is either an I-block or a P-block, the next five blocks will be of the same type. This is easy to describe using the FSM of the specification actor as this actor controls what is allowed to enter the partition.

While these actors are rather simple, it might be easier to describe the behavior of these in some other representation than CAL. A more high-level representation, from which these simple actors can be generated, could be more appropriate. Such representations could also easily be used to decorate the model with information that cannot be expressed in CAL.

Adding Information Software systems are often tested for correctness by adding additional statements which are only used for testing or verifying that the program works according to the specifications. Typically assertions or invariants, pre- and post- conditions, are defined which describe which properties should hold and when. The format of these specification are often a more mathematical format than the language the application is implemented in provides, e.g. the Spin model checker accepts *linear-time temporal logic* statements to describe the correct behavior of the Promela program being verified.

If we return to the example discussed above, where the problem was that the tools cannot know that a I-block cannot be a P-block, it would be practical to add a few statements like $btype \in I \cup P \cup Nvop$ and $I \cap P = \emptyset$, where I and P represent the sets of possible values that belong to these types of blocks. This actually would add a new dimension to the possibilities of specifying the program, first of all, scheduling becomes easier, but second, it is also possible to verify that the program acts according to

the specifications.

8.3 Specification Verification

The point with adding a layer of specifications on top of a dataflow program is to more carefully state the intention of the programmer without the need to make the actual program too explicit or strict. The idea is then that any of these specifications should be removable without the program loosing any functionality. We could actually see such specifications as a description of what is possible in the input stream while the program itself is a description of how the input stream is processed. However, specifications are desired in other places than on the input stream of the program, and if the developer could specify what values are expected on a FIFO or even on a state variable, it would open the possibility of verifying that the implementation corresponds to the intention.

Describing the values that are possible on a FIFO does not seem like a good idea, instead this information belong to the ports connected to the FIFO. This would enable checking if two connected ports are compatible, and either we can trust this information or prove it from the actor representation, then the actors are analyzed for composition. This kind of specification is already used in the Cyclo-Static Dataflow model presented by Wauters et al. in [122]. While this kind of information should not be required for implementing a program, it would introduce interfaces of the actors that first can be verified for the individual actors and then at composition be used to construct an efficient description.

Actor interface compatibility leads us to an approach called Counting Interface Automata (CIA), which was presented by Wandeler et al. [120]. CIA is designed to be a lightweight formalism which can be used to prove behavioral type compatibility between actors of a composition. This is done by capturing the behavior of a CAL program and from this generating an automaton for each actor and one for the framework, meaning the MoC and interconnects of the model. The CIA does in principle describe automata with labeled transitions, where a transition can be an input, output or internal action. The automaton, furthermore, has variables which represent quantitative aspects, which can be used in guards to describe when a transition is enabled. The CIA is used for actor composition, however, with the goal of analyzing the compatibility of actors and not, as in this thesis to remove unnecessary scheduling overheads.

The reason why CIA is mentioned here is that, while this automaton is derived from the CAL representation, it to some extent shows what kind of information we are interested in but, even more importantly, it is an example of a formalism which would be composable with the specifications

of an actor. While this is only speculation of what would be an appropriate representation of additional specification of a dataflow model, the following section is a comparison of some models which have taken different directions to solve related problems. In other words, these are not comparisons of work that is related to the main contributions of this thesis, but a comparison of work that is related to the direction this work should take in the future.

8.4 Related Work

In this chapter, the discussion is more philosophical and for this reason the discussion on related work has to be on the level of the goals and on the ideas of other projects or tools that have a solution or have chosen different direction to solve the problems identified here. For this reason, the comparison of approaches is less technical, and the goal is to highlight the different direction of research within the problem domain.

Canals Part of the scheduling process is to identify how tokens belong together and form larger data units that should be processed together. Such a data unit is then associated with a schedule which triggers the appropriate behavior. Another approach is taken in a dataflow programming language called Canals which was presented by Dahlin et al. in [39]. With Canals scheduling is expressed in the language itself in special, per dataflow subnetwork elements, called schedulers. Both networks and computational kernels are limited to only have one input and one output queue, and instead, the input tokens are grouped into more complex data types. When there is an actual need for several inputs or outputs, special elements called scatters and gathers are used, which define how data types are split or composed between the channels.

In Canals, much of the complexity of the dataflow network is moved to the data types. A network then, when a data token enters, can assign the appropriate schedule for that data type. Also the scheduler is defined in the language, and this means that the developer can create a custom scheduler for a network, or use a more generic scheduler such as a round robin scheduler. If we compare this to CAL, the scheduling in CAL is defined in the actors but not at all on network level while Canals does the opposite by concentrating on the scheduling on network level. For this reason, in a comparison of how to translate CAL actor into Canals, the CAL actors were compared to networks in Canals while CAL actions were compared to kernels [40]; however, it would in that case seem natural to implement the Canals program more coarse grained as the data dependencies between kernels else would result in a massive interconnect.

What seems relevant from Canals in this context is the mapping between

data types and schedules. Even though CAL handles data types on the level of individual numbers (integers etc.), and more complex data types in most cases can be derived for the inputs to a partition, a more explicit representation, at least on specification level would be useful.

PREESM Another direction of the research is to restrict the expressiveness of CAL, to only include statically schedulable actors, in order to allow efficient scheduling on multiprocessor architectures. An example of this is PREESM which is presented by Piat et al. in [104], which uses a restricted version of CAL where actors are implemented according to rules such that each action of an actor must consume and produce an equal amount of tokens on all of the ports. As a result, the CAL network can be translated in to an SDF graph and a balance equation, which then can be used by the PREESM scheduler to, at compile-time generated a multiprocessor schedule.

While SDF is efficient for compile-time scheduling om multiprocessor architectures, SDF does by definition not provide any flexibility and for this reason SDF is not sufficient for many applications. This can to some extent be taken care of by considering the different scenarios, that is, a set of parameters and constraints which specify the conditions under which the application will run [100]. To enable reasoning about scenarios, PREESM uses the Parameterized and Interfaced SDF (πSDF) MoC which was presented by Desnos et al. in [43]. This model extends the SDF model with concepts from PSDF and a MoC called interface-based datalow [103], and introduces explicit parameters and a parameter dependency tree.

The πSDF model includes special elements which are used to explicitly describe the configuration of the model. Such elements are *configuration actors*, *configurable parameters*, *configuration input/output ports*, and *configuration interfaces*. These elements make the parameterization of the model explicit, also through hierarchical levels, and the semantics of πSDF defines the exact behavior and the order in which the steps related to configuration takes place. [43]

This kind of a modeling highlights the control structures by modeling the program as a set of configurable SDF subparts. While the goal of the work related to this thesis is to provide the programmer with a maximal expressiveness, it is certainly relevant to compare these constructs to the control information we try to derive from CAL programs. Also, the conceptual specification actors presented above in Figures 8.1 and 8.2 can be compared to the *configuration actors*.

CAL as Disciplined Dataflow for Analysis as SADF In [112], Siyoum et al. presents an approach where CAL programs are implemented according to the *Disciplined Dataflow Network* (DDN) MoC, which enables

automatic extraction of scenario sequences which are comparable to the static schedules presented in this thesis. The approach identifies all the possible scenarios of a DDN and extracts the SDF graphs corresponding to the scenarios. This enables analysis of the dataflow program, for properties such as deadlock-freedom and boundedness, with known methods. The approach in [112] takes the design of an CAL program in the same direction as was proposed in this chapter by forcing the programs to be implemented according to a set of construction rules making the CAL program conform to DDN.

For a CAL program to conform to DDN, control values, either as input tokens or state variables, must be such that these can only assume a finite number of integer values. DDN uses two types of actors which has different construction rules. The *detector* actors control *kernel* actors by providing the control tokens that decide the firing sequence of the kernel actor. Thus, any control input of a kernel actor must be connected to the output of a detector actor. These control tokens – the output of the detector actors – must be restricted to a finite set of integers. The end result here is rather similar to the scheduling technique presented in Chapter 5; the *detector* actors correspond to the *leader* actor of a partition, and, the restrictions of the control values have a similar effect in both works.

The choice to mention this work in this chapter, and not in the chapters related to scheduling, is due to the fact that this approach is based on specifying the program such that the analysis becomes simple. From the point of view of the discussion in this chapter, this work takes the design of a CAL program in the direction where the intention of the designer is more obvious for the design tools. The difference is that the detector actors actually are part of the program and not simply specifications. However, a CAL program constructed according to DDN, is still simply a CAL program, but with some very useful features. For this reason, a CAL program implemented as DDN should directly by schedulable with the methods presented in this thesis.

Ptolemy It is at this point appropriate to return to the roots of CAL, namely the Ptolemy project [46, 1]. Ptolemy provides a framework for modeling and simulating heterogeneous models, that is, different parts of the model may belong to different MoCs such as SDF or continuous time. Different models are composed with a hierarchically heterogeneous approach, which means that the model is hierarchically structured such that a model can include components with another MoC internally. A Ptolemy model is built from actors, hence the name *actor* in CAL, which can be atomic or composite, depending on if they are built from other actors or if they are at the bottom of the hierarchy. [48]

In the context of dataflow, CAL uses FIFOs for the communication be-

tween the actors, in Ptolemy, however, there are different types of communication, such as mailboxes and rendezvous points, and this is defined by the domain of the composite actor. The domain includes an director, which is concerned with the scheduling of the actors, and receivers, which are concerned with the different types of communication.

We are not, in this context, interested in MoCs such as continuous time, however, Ptolemy also interfaces and specifies different dataflow MoCs, and implements how these different models are composed. It therefore, shows one more way to describe the intention, that is, how the actors are supposed to be fired. Also, the compatibility of these can be checked as already was discussed earlier in this chapter regarding counting interface automata as presented in [120]. This is the essence of what is requested in this chapter, a description of the intended behavior of the domain, a description of the behavior of the actors, and a method for how these can be checked for compatibility.

The three approaches presented above all provide the needed scheduling information by adding special elements to the dataflow network, which describes how the scheduling is supposed or intended to work. It is usually useful to follow research is different communities, and for this we will look at one more work from the world of formal methods.

Rodin and Event-B Rodin is a tool for formal modeling in Event-B, which is a language and a proof method influenced by the B Method [4] and Action Systems [14]. Event-B uses the structure of guarded actions which operate on state variables from Action Systems to describe system behavior, while it uses typed set theory as the mathematical language for defining state structures and events [5]. Compared to CAL, a program in either language is a set of guarded actions, called events in Event-B, which execute as atomic statements and operate on the state variables. The only difference, as CAL is used in dataflow, is that is explicitly defines ports, and the surroundings of the actors describe how these are connected with the FIFOs.

The modeling in Event-B begins from the opposite direction compared to how a CAL program would be implemented. In Event-B the development, or perhaps more correctly, the modeling, starts from more abstract models of the system which describe the intuition regarding what the system should do and what it is allowed to do. The modeling proceeds with something called stepwise refinement, that is, more details are added to the model and the correctness of the refinement, that the refined model does not violate the previous model, is proved by solving some required proof obligations. The proof obligations are automatically generated and often automatically proven. [5]

An Event-B model consists of *contexts* which describe the static part of

the model such as constants and axioms, and *machines* which describe the dynamic part of the model such as state variables and events. The state variables are constrained by invariants which always must hold. The correctness between models and refined models can be proved with the relation between different invariants and axioms, and the proof obligations are generated for this purpose. For the scheduling work presented in this thesis, while the machines resemble the actors, there is nothing similar to the invariants or the contexts in CAL. This is solved in work related to scheduling CAL by using external tools such as model checkers [51] or SMT solvers [124], as discussed in Chapter 5, to prove some properties needed for deciding on how an actor or partition can be scheduled. This again related to the discussion of how specifications could be added to a CAL program.

This makes it easy to describe guards and to prove that these have the specified properties, in Event-B. However, with Event-B, like with CAL, the problem of efficient scheduling or efficient sequentialization is not solved by this, instead it simply helps to prove the completeness of a scheduler. Work regarding scheduling of Event-B has been presented by Boström in [25] and by Degerlund in [41], partly by providing scheduling patterns with their proof obligations, and by allowing processes to fire, up to a maximal number of times before the state variables are updated to the global state. In any case, the scheduling of Event-B resembles the scheduling of CAL, and for this reason it will make sense to more carefully compare the work from both camps.

Chapter 9

Overview of the Original Publications

The second part of this thesis is a collection of pair reviewed published papers that focus on different aspects of the dataflow scheduling problem. The papers are more focused on solving specific problems than the more general first part of the thesis and go more in to detail. The purpose of this chapter is to give an overview of each of the papers and highlight the contribution the papers give to the topic of this thesis. Then for each of the papers, the contribution of the author is also explained.

The papers are presented in chronological order, as they have been published. As can be expected, the first papers present the high level ideas regarding how the scheduling approach can be used and shows some initial proof of concept. The following papers then refine the scheduling approach and evaluates the approach by providing more experimental results. Finally a paper which shows on how the approach can be integrated in a larger too chain is presented.

9.1 Paper I: Scheduling of Dynamic Dataflow Programs with Model Checking

The idea of using a model checker for scheduling a dynamic dataflow program written in CAL originates from the idea that a CAL program simply is a set of communicating state machines. The Promela programming language, which is the input to the Spin model checker, and which basically describes a process network, can then be used to mimic a CAL program and to generate a model checker to analyze the state space of the program. When considering the state space of a program, a schedule is then a path between two program states, while a static schedule is a path between two states that, is always valid when some precondition, in form of a guard ex-

pression, holds. The scheduling is the about identifying these reoccurring states, the guards enabling static schedules, and the actual schedules that link these states.

The main contribution of this paper is to show how the scheduling problem can be expressed as a model checking program. This consists of two parts, first, it is shown how the CAL representation can be translated into Promela, which is the input to the Spin model checker, and an implementation of the Promela code generation that was implemented in the Orcc compiler is presented. Second, the ideas how the schedules are searched for is presented. This includes how the model checker is initialized with an appropriate input sequence corresponding to a type of inputs the program partition accepts, and how to define the state to be searched for as a Linear-time Temporal Logic (LTL) expression, which defines a state where the model checker reports an error if it is found. The concept of using the model checker to produce a trace to such an *error state*, and using the reported trace to describe a schedule is presented in this paper.

This paper presents the initial ideas related to how scheduling of a CAL program can be performed using a model checker. The idea in this paper is that the developer has some knowledge about the application to be scheduled and is able to describe to the model checker what kind of schedules to search for. This is a manual process, where the inputs to a network and the objectives of the schedule search are given by the developer. While the scheduling process mainly is a manual process, this paper shows that the concept works and also, to some extent, what kind of information is needed in order to specify the objective of a model checking schedule search.

The resulting schedule is, already in this paper, described as an FSM, which is important for the future developments as this is a natural way to describe the scheduler of a CAL actor, which is the result of merging a partition after the scheduling stage. The resulting scheduler is, however, not used to merge the actors, but instead, the model checker is used to generate a sequence of C-function calls, that are glued in to the C-code generated from Orcc. This code is then executed and gives some promise of a potential speedup, however, there is only a limited number of experiments.

Author's Contribution: The background and high level idea to this work started from the co-authors of the paper, while the author of this thesis was given the task to realize the idea in a concrete implementation. The task of the author was then to plan what the translation from CAL to Promela should look like, which is described in Section 3, to find out how this can be used to generate schedules, which is described in Section 4, and to implement this in the Orcc compiler. The implementation work, and later the experiments for the paper, was done in cooperation with Ghislain Rouquier. Regarding the paper writing, the author wrote the first draft of

the paper after which the co-authors contributed with improvements and additions.

9.2 Paper II: Scheduling of Dynamic Dataflow Programs Based on State Space Analysis

For the state space exploration to be feasible and hence the model checking of a network of CAL actors, the state of the actors and the actor partition should be described with the minimal information that is required to correctly schedule the program. For an actor, the state is described by the FSM scheduler and the state variables. To reduce the state space, only the state variables relevant for scheduling should be used to describe the actor state. For this, not only the variables that are directly used in a guard are needed but also any variable that are in any sense used to give the value to a variable that is used in a guard. Furthermore, on the level of a network of actors, the cross actor dependencies, that is, every variable inside any of the actions in the partition that is used to produce a value that ends up in a guard in any of the other actors in the partition must also be included.

The main contribution in this second paper is the more careful description of how the model checker is constructed, and which information from the actors is included in the generated Promela code. The paper presents some simple rules for how to create the graph that describes the dependencies between variables in a network of actors. This information both simplifies the description of the actor but also describes the state of a partition such that it can be used for searching for schedules. In practice, the actor is simplified by removing the code that is not relevant, before the code generation. This effects the state space in two ways, first, the variables that simply process data are removed and does not contribute to the state space, second, the code that produces data output is removed which means that data written to queues is always zero valued and only contributes to the state space with the number of tokens. The second benefit from this is that the states of the actors can be described from a pure scheduling point of view, this in turn is important when the schedules to be searched for are to be defined.

Except for this, the paper also present some fresh results from experiments on an MPEG-4 decoder, where the scheduling approach is applied to schedule the motion compensation, IDCT, and texture coding networks independently. The paper then provides results regarding the reduction of guard evaluations needed to schedule the partitions and some measurements of the performance (frame rate) of the program, after manually applying the schedules extracted by the model checker.

Author's Contribution: This paper was a natural continuation of the previous paper, with some of the ideas more clearly defined and with more experiments. The contribution of the author regarding the work for this paper was to develop the rules for describing the partition state more carefully which is presented in Section 3 in the paper and to refine the model checking approach as presented in Section 4.

9.3 Paper III: Static and Quasi-Static Composition of Stream Processing Applications from Dynamic Dataflow Programs

The purpose with any code transformation or optimization is always to improve the run-time performance of an application, be it with respect to speed (frame rate), memory usage, energy consumption, or some other metric. The scheduling and composition of a partition of actors implies that the memory footprint of the program changes and the locality of the code and data will also change. This means that a single metric such as the number of guards needed, is not sufficient for predicting how the performance will change as a result of the composition. Instead, the many parameters regarding the hardware, such as memory architecture, decides how a specific composition will affect the run-time performance.

This paper presents how different types of actors can be composed using different methods, depending on *how dynamic* the actor is. Different parts of a program are then composed in to larger actors, either using a static composition, if the actors are classified to SDF or CSDF, or a quasi-static composition in the other cases. The classification and static composition is here based on [124] while the quasi-static scheduling is performed using the model checking approach. The different configurations of an MPEG-4 decoder program, with different compositions applied and different sizes of the FIFOs between the actors, are used to run experiments on different processors with a set of different videos.

The main contribution of this paper is in the experimental results, which seemingly shows contradicting results but which can be linked to how well the program fits in the cache of the different processors. The paper, compares various aspects of scheduling and composition, and presents results that on the one hand show how different aspects may affect the performance but on the other hand shows the importance of extensive experiments when evaluating an approach like this. The lesson learned from this work is that the platform and perhaps more specifically the cache size and structure, should be used to decide what the schedules should look like and how large the actor compositions should be. In practice it may be hard to directly decide on such parameters from a hardware description, however, these aspects

can still be considered in the design space exploration.

Author's Contribution: This paper presents work related to two scheduling approaches of which the author only has contributed to the other. The contribution of the author in this paper was to write the parts of Sections 3 and 4 related to quasi-static scheduling as well as to plan the experiments and to write the results section (Section 5).

9.4 Paper IV: Modeling Control Tokens for Composition of CAL Actors

A partition scheduler is an FSM where transitions correspond to sequences of actions i.e. schedules. The scheduling decision, that is, the choice between the outgoing transitions in a state is represented as guards of the transitions, where a guard can depend on state variables and input ports. These guards need to be generated or reused from the original actors, and these must be strong enough to correctly schedule the partition. To successfully produce a scheduler for a partition which receives control information from outside the partition, it is essential to have means for reasoning about how the control tokens propagate in the partition in order to decide on whether a set of guards are strong enough to schedule the partition.

This paper presents a different view point of a CAL program, that is, from the point of view of the control tokens. The ideas how to describe the state of a partition presented in Paper 2, are taken further and are formalized, and the resulting dependency graph is simplified to a point when the token propagation of an actor only can have one of a few forms. This makes it easier to reason about control tokens of a partition and to decide what type of proofs are needed in order to show that a set of guards are sufficient for the partition. The different forms of how an actor propagates the control tokens imply different complexity of the verification of the strength of the guards. While in the most simple form, sufficient strength is implied, other types varies from requiring simple proofs to requiring the actor to be transformed. The main idea is that, depending on the constructed dependency graph, the requirements for verification can easily be derived.

The contribution of this paper is that it formally shows how the dependencies of an actor partition is constructed and analyzed. The, almost too, formal description of how to build graphs form a CAL program, shows how each program construct is taken into account. This gives some level of guarantee regarding the completeness of the approach presented.

Author's Contribution: This work was initiated by the author to solve a problem that was identified while working with scheduling of different CAL

programs. The paper is a continuation of the work started in the previous paper, and formalizes the way to view how control information propagates in the program. The first draft was written by the author after which the co-authors contributed with improvements and additions.

9.5 Paper V: High-Performance Programs by Source-Level Merging of RVC-CAL Dataflow Actors

A set of schedules is hardly useful as such, instead, only once the schedule description is used to generate efficient code or to transform a program into a more efficient representation, can some improvement be achieved. The presented scheduling approach produces a scheduler, and a set of schedules in a textual format. This representation then needs to be used to transform the corresponding CAL program according to the composed actors and the schedule describing the execution of this actor composition. We refer to this operation, to create a super actor from a set of actors and the composed actor scheduler, as actor merging. After the actor merging step, a super actor is an actor which has actions corresponding to the sequences of action firings of each schedule, an FSM scheduler corresponding to the FSM that describes the scheduler of the composition, and the ports that communicate with the surrounding actors.

This paper presents source-level actor merging, which means that the merging operation is performed on the level of CAL code. The tools presented takes as input CAL code and a schedule description, and produces a transformed CAL program as output. The difficulty in merging actors is on the one hand to achieve correctness regarding how guards and actor schedulers are generated, and on the other hand, to generate the merged actor such that efficient code can be generated from it. This paper presents formally how the actors are merged and how the different parts of the actors are handled in order to guarantee a correct super actor.

The work in this paper is the basis for the actor merging mentioned in Chapter 7, and which is used to import the scheduler generated by the model checker to the Orcc compiler. The main contribution of this paper with respect to this thesis is that it provides the missing part namely the concrete merging operation, by doing this, it also provides an description of which information needs to be produced by the scheduling framework.

The paper also presents experimental results from a case study of a ZigBee transmitter, run through the full tool chain, including: scheduling, merging, code generation, and running the code on a TTA processor. The experiments show promising results regarding efficient code generation, at least for the class of applications and processors that are studied in this paper. These results show that, using dataflow to describe some applications

on a fine grained level, suitable transformations of the code can produce results comparable to hand optimized programs written in a language like C, however, the dataflow program is easier to transform to fit other architectures.

Author's Contribution: The author's contribution to actually writing this paper was modest, the contributions to the content of the paper is related to Section 4 (especially 4.6) regarding how the scheduler generated from the model checker should be generated in the merged super actor. Apart from this, the actual scheduling work of the experiments presented in Section 5, was performed by the author.

9.6 Positioning the Papers

The papers presented in this chapter and in the second part of this thesis, evaluates and goes in to depth of different parts of the scheduling approach that has been presented in this thesis. None of the papers, however, gives a thorough description of the scheduling problem, which is why the first part contained an extensive description of how scheduling, composition, and actor merging is performed. In other words, the first part is supposed to give the high level idea, and describe the theoretical problems, while the second part, with the papers, concentrates of specific details. Figure 9.1 illustrates how the different papers relates to the different parts of the tool chain.

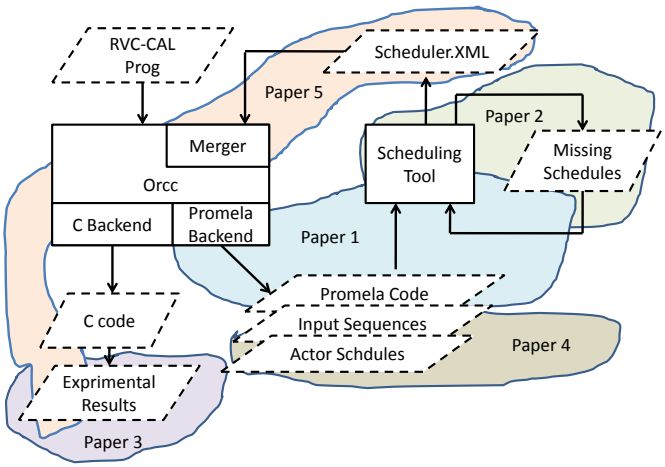

Figure 9.1: Illustration of how the papers correspond to the different parts of the scheduling tool chain.

Chapter 10

Conclusions

This thesis addresses quasi-static actor composition and scheduling, and presents how the scheduling problem can be expressed as a model checking problem. For this to be possible, the actors that form a composite actor are required to be simplified such that a state space analysis is feasible but the behavior is preserved. The presented approach starts from a part of a CAL program which is analyzed and transformed to include only relevant information, and a model checker is produced from this representation. Then a strategy for the scheduling is chosen, typically based on one of the actors in the partition, and the state space is searched in order to find paths between states that partially have been identified at the previous step. Finally, when all necessary paths or schedules have been found, these are translated into a format (based on XML) which is used to transform the CAL program which then is ready for code generation.

The usefulness of the approach depends of the trade-off between potential gain in form of program performance, the effort required from the developer in form of knowledge and time, and simply how well the approach can handle typical actor partitions. Then these properties should be compared to other related approaches in order to find whether the approach excels in any of these directions. Another aspect is how the approach, or perhaps parts of it, has potential to be further developed, and what can be learned from the usefulness of the tool chain that has been constructed for the purpose. This work gives a proof of concept of that this kind of a method is possible to implement and use in a datalow context. This is shown regarding several sub problems, such as, feasibility of state space analysis, identification of control structures, the practical scheduling, and experimental results.

10.1 Retrospective – Goals versus Approach

It is Feasible to Perform State Space Analysis One of the potential problems with using a model checker to analyze a problem for scheduling is the potential size of the state space. In a dataflow program, the state space, if the whole behavior of the program is taken into account, is enormous. If the model would include the data on the FIFOs and the state of the actors, including possible internal data structures of the actors, the state space would not be possible to analyze efficiently. Furthermore, it would be impossible to find a relatively small scheduler if data values also would be part of the state. In the present approach, the state space is significantly reduced both by removing variables which do not affect the scheduling and by treating tokens which carry pure data as simple valueless tokens.

It was shown in the case studies in Chapter 7 that the approach can be used on reasonable large applications, such as a video decoder. However, the efficiency of the scheduling could be significantly improved by methods for finding the minimal buffer sizes (e.g. TURNUS [32]) and by limiting the amount of data that is available on the input queues according to some kind of an approximated balance equation. This tried out manually shows, for the applications in the case studies, that the time spent running the model checker was a few seconds instead of, in the worst case, several minutes when the buffers and inputs were not properly adjusted.

It is, of course, not enough that the program can be simplified such that the state space is manageable. The generated scheduling model must also include enough information such that a complete scheduler can be generated.

Completeness can be verified regarding Guard Strength The correctness of the model is guaranteed by identifying all the required input sequences that result in different schedules and the guards that uniquely corresponds to these. There are two properties that require special attention, the first one is control tokens and the second is independent data paths. Both of these properties are viewed from the point of one actor, which is seen as the leader of the scheduling, and it is then shown that all the other actors in a partition depends on this actor, both regarding token rates and control values.

The control token dependencies can be viewed as a graph describing how control values flow between and through the actors. The graph is then used to decide whether further properties must be proved or if the control structure already is sufficient. The graph also helps the developer to visualize and understand the control structures of an application and, based on this, choose sufficient partitions or in some cases to change the implementation to simplify the control dependencies. In Chapter 5 it was also demonstrated how theorem provers (and model checkers) can be used to further analyze

the control dependencies by proving that the required properties hold for the actors.

In a similar fashion, the token rate dependencies between actors was handled such that it can be shown that the one leader actor can be used to decide the token rates of any other actor that reads inputs of the partition. This is important as else the partition may start to accumulate inputs on one or more of the input ports. This is resolved by constructing a graph of the connections which have deterministic token rates with respect to the leader action, and verify that all inputs depend on the execution of this actor. While this ensures that the partition does not accumulate tokens and by this cause deadlock, this does not guarantee deadlock-freedom. If the partition is part of a feedback loop, deadlock-freedom needs to be verified globally on the schedules of the partition; this is however not discussed in this thesis.

The scheduling decisions are by this a set of firing rules which in turn are guaranteed to decide the behavior of the full partition. With this specification, each firing rule describes a partial input stream and possibly an order in which decisions are to be taken; the complete input streams and the corresponding schedules are searched for in the state space of the partition.

A Few Scheduling Strategies takes us Quite Far The scheduling is based on the possible input sequences and the states the actors may reach processing these. The schedules then are paths between a few of these states such that the schedules can be fired to correspond to any input stream. A scheduling strategy then describes how to identify these states. Often, it is appropriate to describe these states partially, that is, describing some of the properties that should hold in that state. A few strategies were presented in Chapter 5, and these were successfully used to schedule most of the case studies in Chapter 7. It was also shown that the scheduling was automatic, in the sense that the user simply needed to chose the next schedule to search for, for most of the applications, while the once that were not, it could be identified from the analysis of the actor partition that the dependencies will cause problems.

It is difficult to speculate whether the strategies work well because of their generality or because they have been developed with knowledge of the specific case studies. However, choosing one actor to base the scheduling on, and surrounding it with actors which scheduling depends on this actor, basing the scheduling on this actor seems to be an obvious match. The only problem, except from it being difficult to predict the number of states that will be needed for the scheduling, is then predicting which FIFOs may require to store tokens between the schedule firings. This has in this work been left for the developer to decide and can be seen as the potential weakness of the scheduling strategies. The work of the developer could, however,

be simplified by providing an approximation of potential delay tokens or if one input sequence can be predicted to not produce enough tokens for some of the actors during one schedule firing.

The Experimental Results Show Evidence of Usefulness To show the usefulness of quasi-static actor composition techniques, rigorous experiments with different applications, configurations, and platforms, are needed to show that the measured improvements can be generalized. The experimental results presented in this thesis in combination to the results in related approaches show that composition as such is not a guarantee for improved performance, but instead, performance depends on how well the actors fit the target platform. This also shows that designing larger actors to begin with does not make composition unnecessary as different actor sizes are required for portable performance on different platforms.

10.2 This Research in Perspective

Actor composition, possibly together with actor splitting, is essential for dataflow applications to be portable to various platforms regarding performance. While composition only is one step of the design space exploration, it is the one step that enables fitting an application to an architecture with a specific number of cores or with a specific memory hierarchy. As the composition step involves composition of the scheduler, it also decides how much scheduling overhead a program will have.

Another important question is how this approach compares to other approaches for scheduling dynamic dataflow applications. It is quite difficult to compare, and for networks with static actors, it is difficult to compete with approaches such as the classification approach presented by Wipliez et al. in [124] as was shown by the results in Paper 3 [50]. Neither is it fair to compare the exact experimental results with the other approaches that were mentioned before as this would require identical configuration in order to be valid. What, on the other hand, can be compared is the effort it requires from the user to produce the composite scheduler. In the other approaches, the partitioning and scheduling is an automatic step, which does whatever it can before code generation. In the presented approach, however, the developer is part of the process and chooses the partitions, the leader actor, and a strategy for searching for the schedules.

As this approach is more interactive than the others presented, it should be seen more as a part of design space exploration than a compiler optimization. It would, from this point of view, also make sense to use an extended version of the approach to provide the developer with feedback regarding guard strengths and potential partitioning. To take this one step

further, the developer could even provide a library of composition recipes for composite schedulers.

The other alternative to scheduling, as has already been discussed, is to use more restrictive MoCs which provide more predictable scheduling. In any case, even if the MoC provides adequate primitives for describing the control mechanisms, scheduling is still required and similar methods may be required to produce the static schedules for such dynamic actor partitions.

The potential benefit of this work is that it may highlight various design patterns, or perhaps scheduling patterns, which is typically used in dynamic dataflow programs. Simultaneously, it describes the control structures and the potential problems which scheduling some designs. The best way to make use of this contribution is within designing tools that aid the designer either by highlighting dependencies in the design or by allowing the developer to express higher level intentions of the program which can be verified for the actors.

Bibliography

[1] The Ptolemy Project. Department EECS, University of California at Berkeley, (http://ptolemy.eecs.berkeley.edu).

[2] ISO/IEC 23001-4:2009. Information technology - MPEG systems technologies - Part 4: Codec configuration representation, 2009.

[3] A.A.-H. Ab Rahman, S. Casale Brunet, C. Alberti, and M. Mattavelli. Dataflow program analysis and refactoring techniques for design space exploration: Mpeg-4 avc/h.264 decoder implementation case study. In *Design and Architectures for Signal and Image Processing (DASIP), 2013 Conference on*, pages 63–70, 2013.

[4] J.-R. Abrial. *The B-book: Assigning Programs to Meanings.* Cambridge University Press, New York, NY, USA, 1996.

[5] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: An open toolset for modelling and reasoning in event-b. *Int. J. Softw. Tools Technol. Transf.*, 12(6):447–466, November 2010.

[6] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA, 1986.

[7] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Princiles, Techniques, and Tools.* Addison-Wesley, 1986.

[8] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[9] Apple. Grand Central Dispatch. Technical report, 2009.

[10] Arvind and K. P. Gostelow. Some relationships between asynchronous interpreters of a data flow language. In *Formalization of Programming Concepts*, 1977.

[11] Arvind, Kim P. Gostelow, and Wil Plouffe. Indeterminacy, monitors, and dataflow. *SIGOPS Oper. Syst. Rev.*, 11(5):159–169, November 1977.

[12] Krste Asanovic et al. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[13] Russell Atkinson and Carl Hewitt. Synchronization in actor systems. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 267–280, New York, NY, USA, 1977. ACM.

[14] R.J.R. Back. Refinement calculus, part ii: Parallel and reactive programs. In J.W. Bakker, W.-P. Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*, pages 67–93. Springer Berlin Heidelberg, 1990.

[15] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Report on the algorithmic language algol 60. *Commun. ACM*, 3(5):299–314, May 1960.

[16] Lidan Bao, Hongmei Wang, Tiejun Zhang, Donghui Wang, and Chaohuan Hou. Improvement on branch scheduling for vliw architecture. In *ASIC (ASICON), 2011 IEEE 9th International Conference on*, pages 723–726, 2011.

[17] Geoff Barrett. Model checking in practice: the t9000 virtual channel processor. *Software Engineering, IEEE Transactions on*, 21(2):69–78, 1995.

[18] Shuvra S. Battacharyya, Edward A. Lee, and Praveen K. Murthy. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, Norwell, MA, USA, 1996.

[19] Endri Bezati, Richard Thavot, Ghislain Roquier, and Marco Mattavelli. High-level dataflow design of signal processing systems for reconfigurable and multicore heterogeneous platforms. *Journal of Real-Time Image Processing*, pages 1–12, 2013.

[20] B. Bhattacharya and S.S. Bhattacharyya. Parameterized dataflow modeling of dsp systems. In *Acoustics, Speech, and Signal Processing, 2000. ICASSP '00. Proceedings. 2000 IEEE International Conference on*, volume 6, pages 3362–3365 vol.6.

[21] S. S. Bhattacharyya, G. Brebner, J. Eker, J. W. Janneck, M. Mattavelli, and M. Raulet. How to make stream processing more mainstream. In *Proceedings of the Workshop on Streaming Systems*, Lake Como, Italy, November 2008. 3 pages.

[22] Shuvra S. Bhattacharyya, Gordon Brebner, Jörn W. Janneck, Johan Eker, Carl von Platen, Marco Mattavelli, and Mickaël Raulet. Opendf: a dataflow toolset for reconfigurable hardware and multicore systems. *SIGARCH Comput. Archit. News*, 36(5):29–35, June 2009.

[23] G. Bilsen, M. Engels, R. Lauwereins, and J.A. Peperstraete. Cyclo-static data flow. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, volume 5, pages 3255–3258 vol.5, May.

[24] Robert D. Blumofe et al. Cilk: An efficient multithreaded runtime system. In *PPoPP*, pages 207–216, July 1995.

[25] Pontus Boström, Fredrik Degerlund, Kaisa Sere, and Marina Waldén. Concurrent scheduling of event-b models. In John Derrick, Eerke A. Boiten, and Steve Reeves, editors, *Proceedings 15th International Refinement Workshop*, volume 55 of *Electronic Proceedings in Theoretical Computer Science*, page 166–182. Open Publishing Association, 2011.

[26] J. Boutellier, A. Ghazi, O. Silven, and J. Ersfolk. High-performance programs by source-level merging of rvc-cal dataflow actors. In *Signal Processing Systems (SiPS), 2013 IEEE Workshop on*, pages 360–365, 2013.

[27] Jani Boutellier, Mickaël Raulet, and Olli Silvén. Automatic hierarchical discovery of quasi-static schedules of rvc-cal dataflow programs. *Journal of Signal Processing Systems*, 71(1):35–40, 2013.

[28] Jani Boutellier, Olli Silvén, and Mickaël Raulet. Scheduling of cal actor networks based on dynamic code analysis. In *ICASSP*, pages 1609–1612, 2011.

[29] Joseph Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt. Readings in hardware/software co-design. chapter Ptolemy: a framework for simulating and prototyping heterogeneous systems, pages 527–543. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[30] J.T. Buck. Static scheduling and code generation from dynamic dataflow graphs with integer-valued control streams. In *Signals, Systems and Computers, 1994. 1994 Conference Record of the Twenty-*

Eighth Asilomar Conference on, volume 1, pages 508–513 vol.1, Oct-2 Nov.

[31] J.T. Buck and E.A. Lee. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. In *Acoustics, Speech, and Signal Processing, 1993. ICASSP-93., 1993 IEEE International Conference on*, volume 1, pages 429–432 vol.1, April.

[32] S. Casale-Brunet, C. Alberti, M. Mattavelli, and J.W. Janneck. Turnus: A unified dataflow design space exploration framework for heterogeneous parallel systems. In *Design and Architectures for Signal and Image Processing (DASIP), 2013 Conference on*, pages 47–54, 2013.

[33] G. Cedersjö and J.W. Janneck. Toward efficient execution of dataflow actors. In *Signals, Systems and Computers (ASILOMAR), 2012 Conference Record of the Forty Sixth Asilomar Conference on*, pages 1465–1469, 2012.

[34] Gustav Cedersjö and Jörn Janneck. Actor Classification using Actor Machines. In *Signals, Systems and Computers (ASILOMAR), 2013 Conference Record of the Forty Seventh Asilomar Conference on*, 2013.

[35] Donald D. Chamberlin. The "single-assignment" approach to parallel processing. In *Proceedings of the November 16-18, 1971, Fall Joint Computer Conference*, AFIPS '71 (Fall), pages 263–269, New York, NY, USA, 1971. ACM.

[36] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[37] W. D. Clinger. *Foundations of Actor Semantics*. PhD thesis, Massachusetts Institute of Technology, Dept. of Mathematics.

[38] Henk Corporaal. *Microprocessor Architectures from VLIW to TTA*. John Wiley and Sons Ltd, Baffins Lane, Chichester.

[39] A. Dahlin et al. The canals language and its compiler. SCOPES 2009, pages 43–52. ACM.

[40] A. Dahlin, F. Jokhio, J. Lilius, J. Gorin, and M. Raulet. Interfacing and scheduling legacy code within the canals framework. In *Design and Architectures for Signal and Image Processing (DASIP), 2011 Conference on*, pages 1–8, Nov 2011.

[41] Fredrik Degerlund. *Scheduling of Guarded Command Based Models*. PhD thesis, 2012.

[42] JackB. Dennis. First version of a data flow procedure language. In B. Robinet, editor, *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 362–376. Springer Berlin Heidelberg, 1974.

[43] K. Desnos, M. Pelcat, J.-F. Nezan, S.S. Bhattacharyya, and S. Aridhi. Pimm: Parameterized and interfaced dataflow meta-model for mpsocs runtime reconfiguration. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), 2013 International Conference on*, pages 41–48, July 2013.

[44] Edsger W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, 1968.

[45] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975.

[46] J. Eker and J. Janneck. CAL Language Report. Technical Report ERL Technical Memo UCB/ERL M03/48, University of California at Berkeley, December 2003.

[47] J. Eker and J.W. Janneck. Dataflow programming in cal — balancing expressiveness, analyzability, and implementability. In *Signals, Systems and Computers (ASILOMAR), 2012 Conference Record of the Forty Sixth Asilomar Conference on*, pages 1120–1124, 2012.

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

[49] J. Ersfolk, G. Roquier, J. Lilius, and M. Mattavelli. Modeling control tokens for composition of cal actors. In *Design and Architectures for Signal and Image Processing (DASIP), 2013 Conference on*, pages 71–78, 2013.

[50] J. Ersfolk, G. Roquier, W. Lund, M. Mattavelli, and J. Lilius. Static and quasi-static compositions of stream processing applications from dynamic dataflow programs. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 2620–2624, 2013.

[51] Johan Ersfolk, Ghislain Roquier, Fareed Jokhio, Johan Lilius, and Marco Mattavelli. Scheduling of dynamic dataflow programs with model checking. In *IEEE International Workshop on Signal Processing Systems (SiPS)*, 2011.

[52] Johan Ersfolk, Ghislain Roquier, Johan Lilius, and Marco Mattavelli. Scheduling of dynamic dataflow programs based on state space analysis. In *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, pages 1661 –1664, march 2012.

[53] Otto Esko, Pekka Jääskeläinen, Pablo Huerta, Carlos S. de La Lama, Jarmo Takala, and Jose Ignacio Martinez. Customized exposed datapath soft-core design flow with compiler support. In *Proceedings of the 2010 International Conference on Field Programmable Logic and Applications*, FPL '10, pages 217–222, Washington, DC, USA, 2010. IEEE Computer Society.

[54] G. Estrin and R. Turn. Automatic assignment of computations in a variable structure computer system. *Electronic Computers, IEEE Transactions on*, EC-12(6):755–773, 1963.

[55] Joachim Falk, Christian Zebelein, Christian Haubelt, and Jürgen Teich. A rule-based quasi-static scheduling approach for static islands in dynamic dataflow graphs. *ACM Trans. Embed. Comput. Syst.*, 12(3):74:1–74:31, April 2013.

[56] Wan Fokkink. *Introduction to Process Algebra*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 2000.

[57] Martti Forsell. Analysis of transport triggered architectures in general purpose computing. In *Proceedings of the 21st IEEE Norchip Conference.*, Nov 2003.

[58] G.R. Gao, R. Govindarajan, and P. Panangaden. Well-behaved dataflow programs for dsp computation. In *Acoustics, Speech, and Signal Processing, 1992. ICASSP-92., 1992 IEEE International Conference on*, volume 5, pages 561–564 vol.5, 1992.

[59] Marc Geilen, Twan Basten, and Sander Stuijk. Minimising buffer requirements of synchronous dataflow graphs with model checking. In *DAC '05*, 2005.

[60] R. Gu, J. Janneck, M. Raulet, and S. Bhattacharyya. Exploiting statically schedulable regions in dataflow programs. *Journal of Signal Processing Systems*, 63:129–142, 2011.

[61] Zonghua Gu et al. Static scheduling and software synthesis for dataflow graphs with symbolic model-checking. In *RTSS '07*, 2007.

[62] Nan Guan et al. Improving scalability of model-checking for minimizing buffer requirements of synchronous dataflow graphs. In *ASP-DAC '09*, 2009.

[63] J. R Gurd, C. C Kirkham, and I. Watson. The manchester prototype dataflow computer. *Commun. ACM*, 28(1):34–52, January 1985.

[64] Matthew Hennessy. *Algebraic Theory of Processes*. The MIT Press, Cambridge, Mass., 1988.

[65] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS Q.*, 28(1):75–105, March 2004.

[66] Carl Hewitt and Henry G. Baker. Laws for communicating parallel processes. In *IFIP Congress*, pages 987–992, 1977.

[67] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[68] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[69] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[70] Tony Hoare. *Communicating Sequential Processes*. 2004.

[71] Simon Holmbacka, Wictor Lund, Sébastien Lafond, and Johan Lilius. Task migration for dynamic power and performance characteristics on many-core distributed operating systems. In Peter Kilpatrick, Peter Milligan, and Rainer Stotzka, editors, *Proceedings of the 21st International Euromicro Conference on Parallel, Distributed and Network-based Processing*, page 310–317. IEEE Computer society, 2013.

[72] Gerard Holzmann. *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional, first edition, 2003.

[73] J.W. Janneck. A machine model for dataflow actors and its applications. In *Signals, Systems and Computers (ASILOMAR), 2011 Conference Record of the Forty Fifth Asilomar Conference on*, pages 756 –760, nov. 2011.

[74] P. Järvinen. *On research methods*. Opinpaja, 2004.

[75] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, March 2004.

[76] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam.

[77] Richard M. Karp and Raymond E. Miller. Properties of a model for parallel computations: Determinancy, termination, queueing. *SIAM Journal on Applied Mathematics*, 14(6):pp. 1390–1411, 1966.

[78] Richard M. Karp and Raymond E. Miller. Parallel program schemata: A mathematical model for parallel computation. In *SWAT (FOCS)*, pages 55–61, 1967.

[79] E. Lee. Consistency in dataflow graphs. *Parallel and Distributed Systems, IEEE Transactions on*, 2(2):223–235, Apr.

[80] E.A. Lee. Representing and exploiting data parallelism using multidimensional dataflow diagrams. In *Acoustics, Speech, and Signal Processing, 1993. ICASSP-93., 1993 IEEE International Conference on*, volume 1, pages 453–456 vol.1, April.

[81] E.A. Lee and D.G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *Computers, IEEE Transactions on*, C-36(1):24 –35, January 1987.

[82] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[83] E.A. Lee and T.M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773 –801, May 1995.

[84] Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006.

[85] Edward A. Lee and Eleftherios Matsikoudis. A denotational semantics for dataflow with firing. In *Memorandum UCB/ERL M97/ 3, Electronics Research*, 1997.

[86] David J. Lilja. *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, September 2005.

[87] Weichen Liu et al. An efficient technique for analysis of minimal buffer requirements of synchronous dataflow graphs with model checking. In *CODES+ISSS '09*, 2009.

[88] G. Low and S. Huan. Impact of object oriented development on software quality. In *Software Technology and Engineering Practice, 1999. STEP '99. Proceedings*, pages 3–11, 1999.

[89] Christophe Lucarz, Ghislain Roquier, and Marco Mattavelli. High level design space exploration of RVC codec specifications for multi-core heterogeneous platforms. In *Conference on Design and Architectures for Signal and Image Processing, DASIP*, 2010.

[90] T. Malkamaki and S.J. Ovaska. Data centers and energy balance in finland. In *Green Computing Conference (IGCC), 2012 International*, pages 1–6, 2012.

[91] Salvatore T. March and Gerald F. Smith. Design and natural science research on information technology. *Decis. Support Syst.*, 15(4):251–266, December 1995.

[92] R.C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Alan Apt Series. Prentice Hall/Pearson Education, 2003.

[93] Marco Mattavelli, Ihab Amer, and Mickael Raulet. The reconfigurable video coding standard. *IEEE Signal Processing Magazine*, 27(3):157–167, 2010.

[94] Tiffany M. Mintz and James P. Davis. Low-power tradeoffs for mobile computing applications: embedded processors versus custom computing kernels. In *ACM-SE 45: Proceedings of the 45th annual southeast regional conference*, pages 144–149, New York, NY, USA, 2007. ACM Press.

[95] Andrew M. Mironov. Theory of processes. *CoRR*, abs/1009.2259, 2010.

[96] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[97] A. Munshi. OpenCL - Introduction and Overview, 2011.

[98] Y. Neuvo. Cellular phones as embedded systems. In *Solid-State Circuits Conference, 2004. Digest of Technical Papers. ISSCC. 2004 IEEE International*, pages 32–37 Vol.1, 2004.

[99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[100] M. Pelcat, P. Menuet, S. Aridhi, and J. F Nezan. Scalable compile-time scheduler for multi-core architectures. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pages 1552–1555, April 2009.

[101] R. Petersen. *Introductory C With C++*. Surfing Turtle Press, 2006.

[102] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Darmstadt University of Technology, Germany, 1962.

[103] J. Piat, S.S. Bhattacharyya, and M. Raulet. Interface-based hierarchy for synchronous data-flow graphs. In *Signal Processing Systems, 2009. SiPS 2009. IEEE Workshop on*, pages 145–150, Oct 2009.

[104] J. Piat, M. Raulet, M. Pelcat, Pengcheng Mu, and O. Deforges. An extensible framework for fast prototyping of multiprocessor dataflow applications. In *Design and Test Workshop, 2008. IDT 2008. 3rd International*, pages 215–220, Dec 2008.

[105] J.L. Pino, S.S. Bhattacharyya, and E.A. Lee. A hierarchical multiprocessor scheduling system for dsp applications. In *Twenty-Ninth Asilomar Conference on Signals, Systems and Computers, 1995.*, pages 122–126, 1995.

[106] W. Plishker, N. Sane, M. Kiemb, K. Anand, and S.S. Bhattacharyya. Functional dif for rapid prototyping. In *Rapid System Prototyping, 2008. RSP '08. The 19th IEEE/IFIP International Symposium on*, pages 17–23, June 2008.

[107] William Plishker, Nimish Sane, and Shuvra S. Bhattacharyya. A generalized scheduling approach for dynamic dataflow applications. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 111–116, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[108] Ghislain Roquier, Endri Bezati, and Marco Mattavelli. Hardware and software synthesis of heterogeneous systems from dataflow programs. *JECE*, 2012:2:2–2:2, January 2012.

[109] Theo C. Ruys. Optimal scheduling using branch and bound with spin 4.0. In Thomas Ball and Sriram K. Rajamani, editors, *SPIN*, volume 2648 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2003.

[110] Chung-Ching Shen, Shenpei Wu, N. Sane, Hsiang-Huang Wu, W. Plishker, and S.S. Bhattacharyya. Design and synthesis for multimedia systems using the targeted dataflow interchange format. *Multimedia, IEEE Transactions on*, 14(3):630–640, June.

[111] Olli Silvén and Kari Jyrkkä. Observations on power-efficiency trends in mobile communication devices. In *SAMOS*, pages 142–151, 2005.

[112] F. Siyoum, M. Geilen, J. Eker, C. von Platen, and H. Corporaal. Automated extraction of scenario sequences from disciplined dataflow networks. In *Formal Methods and Models for Codesign (MEMOCODE), 2013 Eleventh IEEE/ACM International Conference on*, pages 47–56, 2013.

[113] S. Stuijk, M. Geilen, B. Theelen, and T. Basten. Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In *Embedded Computer Systems (SAMOS), 2011 International Conference on*, pages 404–411, July.

[114] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7), Sep 2005.

[115] B. Theelen, J. Katoen, and Hao Wu. Model checking of scenario-aware dataflow with cadp. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 653–658, March.

[116] 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 *Formal Methods and Models for Co-Design, 2006. MEMOCODE '06. Proceedings. Fourth ACM and IEEE International Conference on*, pages 185–194, July.

[117] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *CC '02*, 2002.

[118] Rob van Glabbeek and Ursula Goltz. *Partial Order Semantics for Refinement of Actions — Neither Necessary nor Always Sufficient but Appropriate when Used with Care.* Centre for Mathematics and Computer Science Note CS-N8901, Amsterdam, The Netherlands, 1989. NewsletterInfo: 36.

[119] D. W. Wall. Limits of instruction-level parallelism. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, volume 26, pages 176–189, New York, NY, 1991. ACM Press.

[120] Ernesto Wandeler, Jorn W. Janneck, Edward A. Lee, and Lothar Thiele. Counting interface automata and their application in static analysis of actor models. In *Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, SEFM '05, pages 106–116, Washington, DC, USA, 2005. IEEE Computer Society.

[121] John Watkinson. *MPEG Handbook*. Butterworth-Heinemann, Newton, MA, USA, 2001.

[122] P. Wauters, M. Engels, R. Lauwereins, and J.A. Peperstraete. Cyclo-dynamic dataflow. In *Parallel and Distributed Processing, 1996. PDP '96. Proceedings of the Fourth Euromicro Workshop on*, pages 319–326, Jan.

[123] M. Wipliez and M. Raulet. Classification and transformation of dynamic dataflow programs. In *Design and Architectures for Signal and Image Processing (DASIP), 2010 Conference on*, pages 303–310, Oct 2010.

[124] Matthieu Wipliez and Mickaël Raulet. Classification of dataflow actors with satisfiability and abstract interpretation. *IJERTCS*, 3(1):49–69, 2012.

[125] Matthieu Wipliez, Ghislain Roquier, and Jean-François Nezan. Software code generation for the rvc-cal language. *Journal of Signal Processing Systems*, 2009.

[126] Dong Hyuk Woo and H.-H.S. Lee. Extending amdahl's law for energy-efficient computing in the many-core era. *Computer*, 41(12):24–31, 2008.

[127] Hervé Yviquel, Emmanuel Casseau, Matthieu Wipliez, and Mickaël Raulet. Efficient multicore scheduling of dataflow process networks. In *Signal Processing Systems (SiPS), 2011 IEEE Workshop on*, pages 198 – 203, Liban, December 2011.

Part II

Original Publications

Paper I

Scheduling of Dynamic Dataflow Programs with Model Checking

Johan Ersfolk, Ghislain Roquier, Fareed Jokhio, Johan Lilius, Marco Mattavelli

Originally published *2011 IEEE Workshop on Signal Processing Systems (SiPS)*. IEEE. October 2011, Beirut, Lebanon.

Paper II

Scheduling of Dynamic Dataflow Programs Based on State Space Analysis

Johan Ersfolk, Ghislain Roquier, Johan Lilius, Marco Mattavelli

Originally published *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing.* IEEE. 2012. pages 1661 –1664. Kyoto, Japan.

Paper III

Static and Quasi-Static Composition of Stream Processing Applications from Dynamic Dataflow Programs

Johan Ersfolk, Ghislain Roquier, Wictor Lund, Marco Mattavelli, Johan Lilius

Originally published *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE. 2013. Vancouver, Canada.

Paper IV

Modeling Control Tokens for Composition of CAL Actors

Johan Ersfolk, Ghislain Roquier, Marco Mattavelli, Johan Lilius

Originally published *DASIP*. ECSI. 2013. Cagliari Italy

©2013 ECSI. Reprinted with permission of ECSI.

Modeling Control Tokens for Composition of CAL Actors

Johan Ersfolk[1,3], Ghislain Roquier[2], Johan Lilius[1], Marco Mattavelli[2]
[1]Åbo Akademi University, Finland
[2]École Polytechnique Fédérale de Lausanne, Switzerland
[3]Turku Centre for Computer Science, Finland

***Abstract*—Dataflow programming is typically used as an intuitive representation for describing multimedia and signal processing applications as computation nodes which communicate through FIFO queues. To run a dataflow network, consisting of several nodes, either run-time or compile-time scheduling is required. Compile-time scheduling techniques are typically based on token rates between nodes and for languages such as CAL, which are expressive enough to describe an actor with any behavior, run-time scheduling is needed in the general case, introducing an overhead. However, the well defined structure of dataflow programs enables analysis of the dependencies of the program and partitions with piecewise static schedules can be derived. In this paper we describe how actor partitions with control tokens can be modeled such that a correct scheduler, where most scheduling decisions are taken at compile-time, can be derived for the resulting composed actor.**

***Index Terms*—Dataflow programming, actor composition, MPEG-4 decoder**

I. Introduction

The typical applications for dataflow programming are from the domain of multimedia and signal processing. Signal processing algorithms can often be described as functional mappings between the inputs and the outputs and can be implemented with static token rates. Multimedia applications on the contrary, use various coding tools (i.e. algorithmic blocks) and allow different combinations of coding tools to be used for the processing. When implementing such applications as dataflow programs, this means that, while many coding tools such as various transforms can be implemented as synchronous dataflow (SDF) [1], a well-known static dataflow model, the programs implementing multimedia applications will include choices that must be taken at run-time. These choices are a result of the need to choose the appropriate coding tools but also, in some cases, a result of implementation choices in the dataflow program.

The RVC-CAL actor language has been standardized as the language to be used to describe the different coding tools of the MPEG Reconfigurable Video Coding (RVC) standard [2]. RVC-CAL provides the expressiveness that is needed to describe the various coding tools. The coding tools, represented by RVC-CAL actors, are assembled into a program by connecting the actors with unidirectional order preserving queues. The actors are allowed to execute in parallel and define their internal scheduling depending on the actors inputs and/or internal state. As a consequence, compile-time scheduling of an RVC-CAL program, consisting of a network of connected actors, is not a trivial task. Scheduling in an SDF like manner based on token rates at compile-time is in general not an option as the actors does not have static token rates, instead the token rates may depend on the input values of the actor or on the current state of the actor. Dynamic (run-time) scheduling introduces significant overhead from evaluating guards at run-time, for this reason, quasi-static scheduling methods, where piecewise static schedules are identified at compile-time while a minimum number of scheduling decisions are left for run-time, have been proposed [3], [4], [5].

Dynamic scheduling is needed when actors have input dependent conditions and consequently, some of the queues carry control tokens between the actors. The actors in turn can retransmit control tokens by defining how an output port depends on an input port. In this paper we focus on constructing models that capture this essential information which is needed for scheduling and also to decide on how to partition a network such that the scheduling becomes efficient. The abstract model of the control token propagation enables reasoning about whether a partition after a composition still accepts every input sequence allowed in the original program. The problem is related to the actual control values entering the partition and thereby also describes the guards required to perform the scheduling. When the models are in place, approaches such as [5] can be used to find the actual schedules.

II. Background and Related Work

A CAL program consist of a set of actors exchanging data tokens through First-In First-Out queues (FIFOs). The actors execute the program by firing eligible actions. Actions are eligible depending on the availability of input tokens, the values of the input tokens, and the internal state of the actor (inside guard statements). Each action may consume and/or produce tokens from one or more input or output ports connected to the FIFO channels; an action may also have no input or output.

The execution of a CAL dataflow program is asynchronous (i.e. it abstracts from time) and each actor can fire independently from all the others as far as one of its actions is eligible. However, sometimes the number of processing elements is lower than the number of actors. In that case, actors assigned on the same processing element must be scheduled by an external scheduler. An actor can fire an action only

if one of its actions is eligible. An action is eligible if: 1. tokens are available, 2. its guard expression (including any state predicate) evaluates to true, 3. it is enabled by the action scheduler, and 4. it has a higher priority if more than one action is enabled by the action scheduler in that state. The CAL action scheduler, if present, is a CAL language operator that expresses, in the form of finite state machine (FSM) transitions, when actions are eligible. An action is only fired if the current state has a transition corresponding to that action.

The scheduling of a CAL program is distributed on each of the actors and is described by the FSM, guards and priorities. In a software implementation, where there is a smaller number of processor cores than actors or the actors are too fine grained for the architecture, actors mapped to the same core must be scheduled.

Different quasi-static scheduling approaches, where static schedules of parts of a CAL program can be derived, have been presented. In [4] CAL actors are classified to belong to more restricted models of computations such SDF [1], CSDF [6] or PSDF [7], in order to allow more efficient scheduling. In [8] static regions, spanning over several actors, are identified and scheduled at compile-time. Other approaches such as [5] and [3], make use of the network partition state, consisting of the values of a subset of the variables or inputs to the partition, to find deterministic sequences of action firings. For such approaches it is of importance to identify the information in a network that is of relevance for the scheduling of the program in order to choose partitions and find the possible scenarios of those partitions. Another approach, presented in [9] constructs a model, called a machine model, which captures the action selection process of the actors and can be used to reduce the number of possible execution paths and remove the evaluation of unnecessary conditions. This model also enables actor composition based on token counts.

The difference in our approach is that we attempt to model how actors interchange actual control values and not only the number of tokens. The idea is to make control tokens a visible part of the model such that one or more of the scheduling approaches mentioned above can be used for the actual static scheduling.

III. Control Token Propagation

In a CAL actor, the token rates are fixed for each action. For simple actors, it is therefore possible to describe the behavior of individual actors as more restricted models of computation with predictable token rates [4]. However, the existence of control tokens makes actor composition based on token rates impossible in the general case. While the token rates of actions are explicitly defined, the propagation of control tokens must be derived from the implementation of the actions. A value on a FIFO is considered to be a control token if it will eventually end up in a guard expression; the path of this control token is analyzed backwards through the network, from the guard towards the input-stream. This search either terminates at an actor sending the value of a constant or a variable not depending on inputs, or at the input stream. In any case, this control token path describes what must be known in order to schedule the program and where this information is generated.

e	$\in$	Expressions
S	$\in$	Statements
x, y	$\in$	Variables
n	$\in$	Numerals
op	$\in$	Operators
p, q	$\in$	Ports
$id, state$	$\in$	Identifiers
e	$::=$	$x \mid n \mid e_1\ op\ e_2$
S	$::=$	x **:=** $e \mid skip \mid S_1;\ S_2$
		$\mid$ **if** $(e)\ S_1$ **else** $S_2 \mid$ **while** (e) **do** S
actor	$::=$	**actor** *id* () **... ⇒ ...**
		vars
		action
		schedule fsm *state* : *trans* **end**
		end
action	$::=$	**action** : *id* : *ports* ⇒ *ports*
		vars
		guard *e*
		do *S* **end**
		end
		$\mid$ *action*; *action*
vars	$::=$	x **:=** $n \mid$ *vars*; *vars*
ports	$::=$	p**:** [x] $\mid$ *ports*; *ports*
trans	$::=$	*state* (*id*) → *state*
		$\mid$ *trans*; *trans*

TABLE I
THE BASIC SYNTAX OF AN ACTOR DECLARATION

$Network$	$=$	$(Actors, Ports, \chi)$
p, q	$\in$	$P_i \cup P_o$
χ	$:$	$P_i \nrightarrow P_o$

TABLE II
NETWORK STRUCTURE

In [10] a method for identifying the control token paths by a compiler is presented and used to find the information that is required for scheduling the program. The analysis is used to find the variables and operations that have an impact on scheduling while the other variables and operations are removed from the model. By removing any variable not related to scheduling, the state space of the program is reduced to make the state space analysis feasible. The resulting model is then used for extracting quasi-static schedules by using the model checking technique presented in [5].

In order to construct a correct scheduling model to be used for actor composition, the control tokens must be modeled such that it is possible to verify that the guards used for a composed actor cover all possible inputs of the composition. To illustrate the approach we will use the, slightly simplified, CAL syntax presented in Table I.

A. Dataflow Analysis (at Instruction Level)

The information we are interested in is the relationship between guards, variables, and ports. We have both global and local variables, $V = V_g \cup V_l$, where global variables mean state variables of the actor and local means variables local to the action that are used to perform the calculation. The actors also have input and output ports, $P = P_i \cup P_o$ and for simplicity, we consider ports to be a special type of variables $P \subseteq V$ and these correspond to local variables according to the patterns specified in the actions (as p:[x] in Table I).

The dependencies between variables in an actor can be described as a binary relation, $R \subseteq V \times V$, which is a subset of the pairs of variables in the actor such that $(x,y) \in R$ indicates that variable x depends on variable y. The relation is built according to the rules in Table III for each actor. For each action of an actor we add to R the variable pairs resulting from the action code and the variables accessed by guards as pairs of guards and variables. We use a special set of variables, $grd_{action} \in V$, to represent the result of evaluating a guard and to indicate that a variable is used in a guard. Table III describes the generation of R from the actors as an annotated type system where a judgement of the form $S : \Sigma \xrightarrow{V}_{R} \Sigma$ indicates that statement S modifying the program state between two program states in Σ modifies the variables in V and produces the relations R.

For a single actor, the set of variables used for scheduling can be described as $V_S = \{x \in V \mid (grd, x) \in R^+\}$, where R^+ is the transitive closure of R. For a partition with more than one actor, this is not enough as we also need to take into account the scheduling information passed between actors. To do this, we identify, for each actor, the set of ports used for scheduling $P_g = P \cap V_S$ and use the relation χ from the network description, which describes how ports are connected in the dataflow network, to find the output ports connected to these input ports and add $\{p \in P_o \mid q \in P_i \cap P_g \wedge (q,p) \in \chi\}$ to P_g. These newly added output ports may in turn depend on input ports and P_g must be updated to include $\{q \in P_i \mid (p,q) \in R^+ \wedge p \in P_g\}$ to P_g. We need to iterate these two steps until no new ports are added and finally add the resulting scheduling variables $V'_S = V_S \cup \{x \in V \mid (p,x) \in R^+ \wedge p \in P_g \cap P_o\}$.

The next step is then to transform R into something that gives a simple representation of the variable dependency of the network partition. We simply want to describe the state variables that either a guard or a control output port depends on. The exact behavior is not required and is unnecessarily complex, what is actually needed is the variables and the abstract behavior of these variables. There are only a few types of possible behavior: a guard (or control port) may depend on an input port or on variables, these variables may or may not depend on input ports and may or may not have memory by depending on themselves. We introduce another relation $R_S = \{(x,y) \in R^+ \mid x \in C \vee (x = y \vee y \in P_i) \wedge \exists z \in C, (z,x) \in R^+\}$ where $C = P_g \cap P_o \cup \{grd\}$ to represent the control value dependencies for the current partition, this is a partition specific relation which means that it may not be valid if an actor is added to or removed from the partition.

$$[var]\ x : \{x\}$$

$$[num]\ n : \varnothing$$

$$[op]\ \frac{e_1 : V_1 \quad e_2 : V_2}{e_1 \text{ op } e_2 : V_1 \cup V_2}$$

$$[ass]\ \frac{e : V}{x := e : \Sigma \xrightarrow[\{x\} \times V]{\{x\}} \Sigma}$$

$$[skip]\ \ skip : \ \Sigma \xrightarrow[\{\varnothing\}]{\{\varnothing\}} \Sigma$$

$$[seq]\ \frac{S_1 : \Sigma \xrightarrow[R_1]{V_1} \Sigma \qquad S_2 : \Sigma \xrightarrow[R_2]{V_2} \Sigma}{S_1;\ S_2 : \ \Sigma \xrightarrow[R_1 \cup R_2]{V_1 \cup V_2} \Sigma}$$

$$[if]\ \frac{e : V_c \quad S_1 : \Sigma \xrightarrow[R_1]{V_1} \Sigma \quad S_2 : \Sigma \xrightarrow[R_2]{V_2} \Sigma}{\text{if } (e)\ S_1 \text{ else } S_2 : \Sigma \xrightarrow[R_1 \cup R_2 \cup ((V_1 \cup V_2) \times V_c)]{V_1 \cup V_2} \Sigma}$$

$$[wh]\ \frac{e : V_c \quad S_1 : \Sigma \xrightarrow[R]{V} \Sigma}{\text{while } (e) \text{ do } S_1 : \Sigma \xrightarrow[R \cup (V \times V_c)]{V} \Sigma}$$

$$[action]\ \frac{e : V_g \quad S_1 : \Sigma \xrightarrow[R]{V} \Sigma}{\text{action grd } e \text{ do } S_1 : \Sigma \xrightarrow[(\{grd\} \times V_g) \cup R]{V_g \cup V} \Sigma}$$

$$[a\_seq]\ \frac{action_1 : \Sigma \xrightarrow[R_1]{V_1} \Sigma \quad action_2 : \Sigma \xrightarrow[R_2]{V_2} \Sigma}{action_1;\ action_2 : \Sigma \xrightarrow[R_1 \cup R_2]{V_1 \cup V_2} \Sigma}$$

$$[actor]\ \frac{action : \Sigma \xrightarrow[R]{V} \Sigma}{\text{actor ... } action \text{... end} : R}$$

TABLE III
BUILDING VARIABLE DEPENDENCY RELATION R OF AN ACTOR. THE JUDGMENTS (ARROWS) INDICATES WHICH VARIABLES V ARE ASSIGNED AND WHICH RELATIONS R ARE PRODUCED.

IV. Actor Composition with Control Token Dependency

We use the information generated in the previous section to decide how actors can be composed. The actor compositions of interest is any two actors where one produces a control token which is used by the other actor. The idea is to compose actors with redundant scheduling and for this the scheduler (guards, FSM, priority) of the front actor is used to schedule the composition. The requirement for this to be possible, is that, a specific firing sequence in the front actor implies a specific firing sequence in the second actor. When this is not the case, and the guards does not completely describe the behavior of the composition, we either must transform the front actor to have appropriate guards or not compose the two actors.

The state of an actor is described by the state variables in V_S, and the states of the actor FSM. The scheduling states $\Sigma_S = \{\sigma_1, \sigma_2, ..., \sigma_M\} \subset \Sigma$ is the smallest possible subset of the set of program states, initially including the initial state and the states with input dependent transitions, of the front actor. The generated scheduler is a state machine

$$grd \prec a_1 \prec a_2 \prec a_1 \prec a_2 \prec ..$$
$$\Downarrow$$
$$grd \prec b_1 \prec b_2 \prec b_1 \prec b_2 \prec ..$$

Fig. 1. Action a_2 of the front actor sends an control value that enables a guarded sequence in the second actor, the scheduling of the composed actor is then about interleaving the actions from the two actors after this point.

where the states correspond to the program states Σ_S and the transitions correspond to firing a sequence of actions i.e. a schedule. We define $S_A = \{s_1, s_2, s_3, ..\}$ to be the set of action firing sequences between scheduling states in actor A and $s_i = a_1 \prec a_2 \prec ... \prec a_N$ to be a schedule where a_1 is fired before a_2 etc. For a composition, each state in Σ_S corresponds to some specific value of the variables in V_S and the FSM of the second actor. When a scheduling state is reached with other values on these, we need to add a new state to Σ_S as this state may require slightly different schedules.

A. Control Token Graph and Guard Expressions

We will consider the composition of two actors where the first actor produces a control token consumed by the other. Given two actors, the objective is to find to what extent the scheduling in these is redundant. The two actors can be described as a set of actions such that $A, B \subseteq \{a, b, c, ...\}$. The control tokens produced or consumed are defined as $T \subseteq \mathbb{Z}$. Actor A is the front actor of the composition and it produces a control value for actor B, the objective is to check if firing a control token generating action in A implies that a specific action will become enabled in B (see Figure 1).

We define two relations, $O_A \subseteq A \times T$ and $I_B \subseteq T \times B$ where, O_A is the relation from actions to possible output tokens for actor A and I_B is the function [1] from potential input tokens to actions with a guard accepting this token in actor B. We also define the functions $grd_a(t) \in \{true, false\}$ and $body_a(t) = t'$ representing the guard and calculations in action a where $t, t' \in T$ are the input and output tokens.

We have a relation between the control token production/consumption in the two actors $f : A \times B$ such that $f = \{(a,b) \in A \times B \mid (a,t) \in O_a \wedge (t,b) \in I_b \wedge t \in T\}$. Alternatively we can say that f is the composition of the two relations $f = O_A \circ I_B$. If f is functional, that is, each element in A maps to an unique element in B, then the scheduling of A can be used to schedule B. We can derive the circumstances when there is a functional relation between the schedule in the two actors. To make the discussion easier to follow, the action that produces a control token can be put in one of the groups shown in Figure 2.

We can decide how the control token is produced by actor A, simply by performing some checks on the variable relation R_S and we simply check from which input ports and state variables the control value is generated and whether the state variables in V_S depend on input ports or themselves. The result is the information about if we know the actual values of the control tokens that can be generated or only some properties described by a guard in actor A. For actor B which receives the control token, the control token is either used directly in a guard or passed through one or more variables before ending up in a guard. When the control token passes through variables in actor B, it becomes necessary to prove that we know every possible value of the control token as these variables are used to describe the scheduler state.

[1] We require that I_B is a function, that is, one input value will only enable one action in the specific state. O_A on the other hand is not necessarily a function as an action can produce outputs with different values.

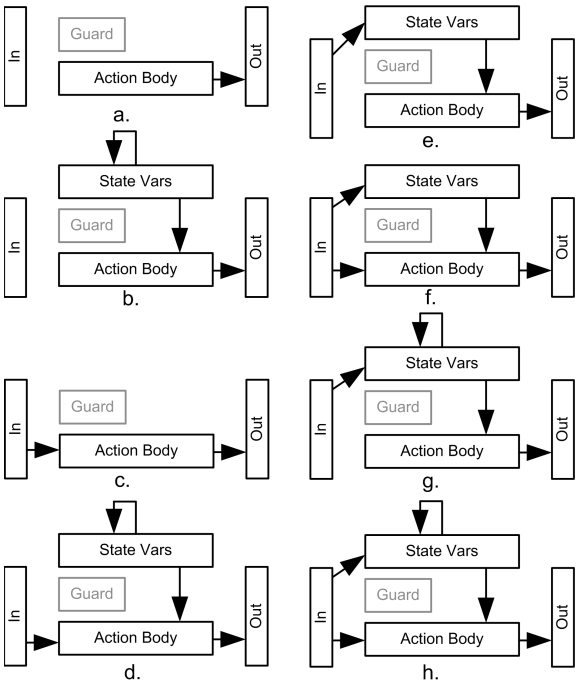

Fig. 2. Possible paths a control token can take through an action: a) generated within the action, i.e. a constant, b) depending on a counter, c) generated from an input value, d) a combination of a counter and an input value, e) generated from previous input values stored in state variables, f) depending on current and previous inputs, and g) depending on a variable with memory of previous inputs.

1) When an Actor is the source of the Control Token: In the first two cases (in Figure 2) the actor is the source of the control token as it does not depend on input in any sense. Case (a) describes the — from a scheduling point of view — best situation, where the action outputs a constant, meaning that the action always produces the same output value. This situation occurs when R_S does not contain a pair connecting the output port to either an input port or a state variable which depend on itself.

$$\forall (x,y) \in R_S : \left[x \in P_g \cap P_o \Rightarrow y \notin P_i \vee (y,y) \notin R_S\right] \quad (1)$$

In the second case (b), there is a variable $(y,y) \in R_S$, this means that the variable updates its value as a function of itself and typically is a counter.

In both cases the control token t' is generated from variables which does not depend on the input t. As there is no input

dependency, each of the variables, related to generating this control value, is seen as having a known value in that state $\sigma_s \in \Sigma_S$ and therefore the output is generated from a set of constants. This means that $O_A = \{(a,t') \mid grd_a(t) = true \wedge t' = body_a(\sigma_s)\}$, and O_A is clearly a function for the state σ_s. As I_b is required to be a function, $f = O_A \circ I_B$ is also a function and A makes the scheduling of B redundant. The actor A is in both cases the actual source of the control value and the generated control values can be determined from the sequence of actions fired. For case b, where there is a counting variable, the scheduler may get more states if there is a scheduling state in actor A which can be reached with different values of the counting variable.

For the second actor, receiving the control token, we know that we cover every possible control token, and for this reason there are no restriction on how this token is used (state variables etc.). Having a counter variable in the receiving actor, of course, may lead to many scheduler states but will not affect the correctness.

2) When Control Token Passes Through Actor: For the following two cases (c and d), the output control value depends on an input port and cannot be derived from the actor only. In case (d) the control token is a combination of the counter and the input value which means that the state of the actor also affects the output value. For a specific state σ_s, including the counter variable, the generated output value is a function of the input and a specific input value always corresponds to a specific output value for a specific state σ_s. This situation is found if R_S connects the control output port to an input port but not through a state variable.

$$\begin{aligned} &\exists (p,q) \in R_S : p \in P_g \cap P_o \wedge q \in P_i \wedge \\ &\quad \neg \exists x \in V_S : (p,x) \in R_S \wedge (x,q) \in R_S \end{aligned} \tag{2}$$

Here, it is not always the case that the behavior of actor B can be derived from the behavior of actor A. For the composition to make sense, the control token must be used in a guard in actor A before actor B uses it in a guard, this is implied if the action sending the control token have a guard using the input control token (through a peek). The generated control tokens can then be expressed as $O_A = \{(a,t') \in A \times T \mid grd_a(t) = true \wedge t' = body_a(\sigma_s,t)\}$, if $t' = t$ this case also corresponds to two actors sharing the same input token. Now, O_a is not necessarily a function; if grd_a accepts more than one value then O_a may contain several output values for one action. It can still be possible to show that f is functional by showing that Proposition 3 holds.

$$\forall (a,t_1),(a,t_2) \in O_A : \left[(t_1,b),(t_2,c) \in I_B \Rightarrow b = c\right] \tag{3}$$

We also get more restrictions for the second actor. If the second actor has a guard directly reading the control token from the input port (peek) and there is a functional dependency between the guards in the two actors, the actors can be composed. If the second actor reads the control value to a variable before using it in a guard, this variable is also used to describe the scheduler state and may introduce new states in Σ_S. The problem is that the guard often accepts a wide range of values and the program behavior should be analysed for each of these.

3) When there are Input Dependent Control Variables: The last four cases have one thing in common - the control token is produced from an input port and passes through a state variable.

$$\exists (p,x),(y,q) \in R_S : \left[p,q \in P_g \wedge x = y\right] \tag{4}$$

For a composition using the guards of the front actor to be possible, either the action setting the value of the variable from the input port or the action setting the value of the output port from the variable must have a guards to be compared with the guard in the receiving actor. To resolve these cases, it is necessary to analyze the order in which these variables are read and written and to check at what point there is a guard using this value. An alternative solution is to remove the variable from the scheduler state and instead use its value in the same fashion as an input port. The strategy for scheduling each of these cases is to, if possible, transform the actor to resemble case c and then perform the analysis accordingly.

Case e may correspond to case c if the action a_1 that sets the variable from the port precedes the action a_2 that writes the control token, $\forall s \in S_A : b \in s \Rightarrow a \in s \wedge a \prec b$, if these actions are always fired in sequence, they could be merged and correspond to case c. If this is not the case, the state variable must be seen as an input port and the schedule with the action producing the control token must have a guard depending on this variable. In this case, the same rules as for case c applies.

In the sixth case (f) the output is a combination of current and previous inputs. As we cannot assume that we know every possible input that can be read to the state variable, the state variable must be seen as an input port in the same fashion as case e, then for the guard using this variable, we can apply the same rules as for case c. The same applies for cases g and h, where the state variable depends on input but also on the history of (potentially all) previous inputs. The variable is likely to have many states and should not be part of the scheduler state, instead we can consider the variable to act as an input port, but only in the case the front actor uses this variable in a guard which either is the same as the one producing the control or precedes it in each schedule.

B. Partitioning, Composition, and Scheduling

From the discussion above it is clear that the scheduling of some compositions with control tokens requires much less work from the compiler or design tools than other. What can easily be identified are the actors which are the source of the control token. When, for some reason, a partitioning where the front actor is not the source of the control token is requested, it may still be possible to show that there is a functional relation between the actions in the actors making the composition feasible. Otherwise, searching *upstream* for the actor with the source and performing that composition first, may resolve the problem. Figure 3 describes the different

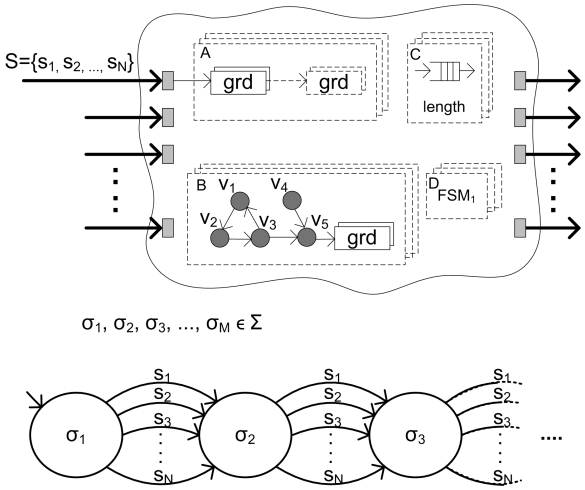

Fig. 3. The state of the partitions is defined by state variables, FSMs and FIFO states. The scenarios of the partitons are decided by the input sequences S. The number of schedules needed is $M \times N$ where M is the number of scheduling states and N is the number of input sequences.

aspects of a partition constructed for composition, here A describes the control tokens entering a partition and how these may be propagated inside the partition. For the scheduling of a partition, it is these guards that need to be analyzed as described above.

While there are restrictions on control values entering a partition from the outside, control tokens with dependencies inside the partition only, can not affect the correctness of the model and therefore we *can* allow any type of variable dependency inside the partition. This kind of control structure is illustrated as B in Figure 3 where a guard depends on several variables, possibly from different actors, but does not depend on the partition inputs. Depending on the structure of the variable dependency, some variables, where there is a cyclic dependency, can hold many values and cause the partition scheduler to have many states.

A scheduler for a partiton can be described as the FSM in Figure 3 where each state has an outgoing transiton corresponding to the possible input sequences of the partition. How the transitions connect the states depend on what the program state is after the corresponding schedule and does not necessarily follow the example in Figure 3. The number of states needed in the FSM depends on the number of states the rest of the partition will end up in when the actor used as front actor executes between its scheduling states. The number of schedules needed is the number of states times the number of alternative transitions in the scheduling states.

V. Case Study and Experimental Results

The presented analysis is implemented as a set of compiler passes generating graphs describing scheduling dependencies. The idea is that the compiler can use this information to decide how to partition the program and whether to compose some actors into a larger actor or run these separately; also the programmer could use this information to find and remove

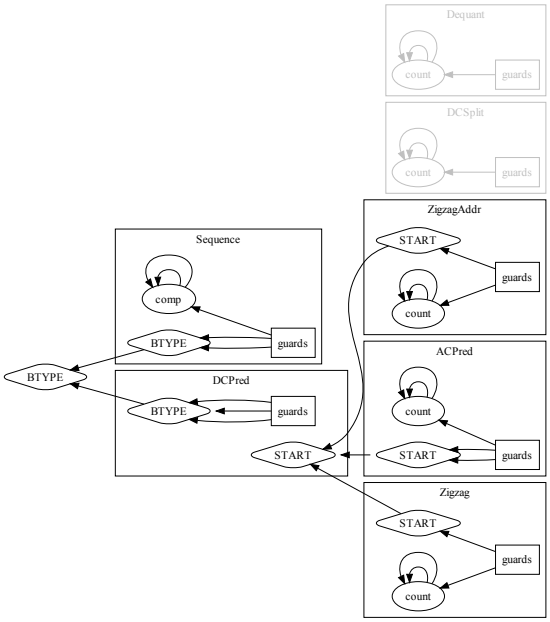

Fig. 5. An abstract dependency graph for the intra prediction network of an MPEG-4 decoder. Circles represent state variables, diamonds represent ports.

Scenario	Port	read/write
start	BTYPE	1 (2048) + 2 (data)
	START	1 (constant)
inter_ac	BTYPE	1 (514)
	START	1 (constant)
other	BTYPE	1 (0)
	START	1 (constant)
intra	BTYPE	1 (1024)
	START	1 (constant in if-statement)

TABLE IV
THE DIFFERENT SCENARIONS OF THE *DCPred* ACTOR. THE TABLE SHOWS THE ACTION STARTING A SCENARIO, THE CONTROL PORT AFFECTED AND THE PATTERN OF THIS PORT IN THE SPECIFIC SCENARIO.

unnecessary dependencies from the program or simply to create partitions manually.

To demonstrate the approach we will use the texture processing part of an MPEG-4 decoder as example. The control value graph shown in Figure 5 gives a graphical view of the relation R_S, for the decoder sub-network, where ovals and diamonds represent variables and ports respectively and arrows indicate that there is a dependency in R_S.

The graph of this sub-network shows three different types of dependencies between actors, we will investigate how the actor *DCPred*, which according to the graph has control dependencies to several of the other actors of the network, can be used to schedule the partition. This actor depends on an input value from outside the partition, shares this same input value with one of the actors (*Sequence*) and produces the control tokens for three of the actors. Two actors do not depend on any values from other actors.

The first dependency to be resolved is the one between the two actors sharing the control value from outside the partition and to find the guards that describes the behavior of these.

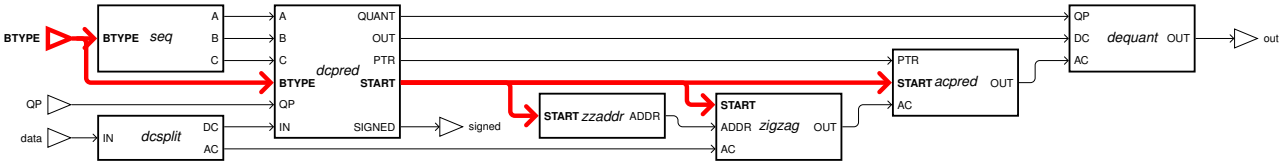

Fig. 4. The intra prediction network of a RVC-CAL MPEG-4 decoder, with the queues carrying control tokens high-lighted (red color).

In practice this means that we must show that the guards of these actors are compatible; according to Proposition 3 in Section IV-A there must be, for each token accepted by one of the guards in actor A, one specific guard in actor B that accepts these tokens. The idea is to use a similar technique as in [4] to check this property, in this case study it was done by hand to give an example of the property. Figure 6 describes the resulting relation, it shows that the relation is functional as for each accepted input in *DCPred*, there is a specific guard in *Sequence* that accepts this token. This means that we can use the guards of *DCPred* to schedule *Sequence* but note that the opposite would not work.

The second dependency to be resolved is the control value sent from *DCPred* to three of the other actors in the sub-network. The graph in Figure 5 shows that the control value is produced from constants and does not depend on input nor on the state of the actor, Table IV, however, shows that the *intra* scenario chooses one of several possible constants inside an if-statement. In the case of an if-statement, the options are to either keep a dependency to the condition and include it in the model or remove this dependency and check how the different constants affect the scheduling. In this case, as this if-statement depends on input values, we remove it and analyze the different possible cases.

We know that the output value is generated from constants and that there will be one value from each case in the if-statement, the values of the outputs can therefore be found, simply by executing each of the branches. Figure 7 shows the relation between the scenarios in *DCPred*, the output values, and the scenarios in *ACPred*; the relation to the other two actors using this control value is omitted as these are similar to this one. According to Section IV-A the relation between the guards of the scenarios must be functional for the scheduling of the second actor to be redundant. For Figure 7, we can see that while the relation from *DCPred* to the output value is not functional, the relation from *DCPred* to *ACPred* is functional as the guard of *start* in *ACPred* accepts each of the values generated by *intra* in *DCPred*.

The result of the analysis of this specific network, is, that the scheduler (guards, FSM) of the *DCPred* actor completely describes how to schedule the rest of this partition. The actors of this partition can therefore be composed into one single actor, the benefits of this is that every scheduling decision of the other actors than *DCPred* will be removed and the FIFOs inside the composition can be replaced with variables/arrays as soon as the action firing sequences have been specified by methods such as [5].

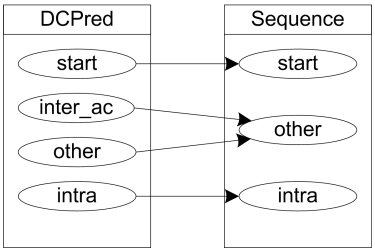

Fig. 6. The relation between the guards of actions triggering each of the scenarios in the two actors sharing the control input value of the sub-network.

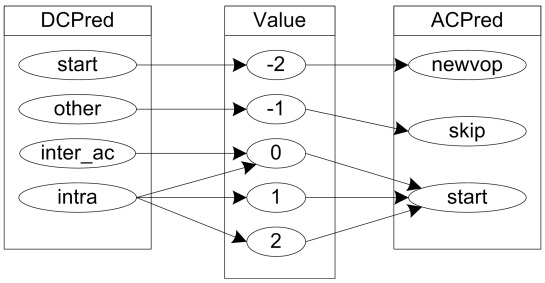

Fig. 7. The relations between the scenario run in *DCPred*, the produced control value and the guards accepting these control values in *ACPred*.

A. Resulting Scheduler

The scheduler for the composed network partition can be based on the *DCPred* actor. This actor has one FSM state where the actions depend on input and these actions describes the different types of input the network can process. From this we know that the scheduler is an FSM with at least one state and four transitions, corresponding to schedules leaving that state. If it is not possible to construct each schedule such that it consumes the inputs and runs the partition to a state where the FSMs and scheduling state variables V_S corresponds to the initial state, more states are needed to describe the scheduling of the partition.

For the actual scheduling, the variables in Figure 5 and the FSM states of the actors define the state of the partition. The internal FIFOs of the partition are required to be empty after a schedule has terminated. Using the model-checking technique in [5], we generate a model-checker using the Promela backend of the Orcc compiler and provide input accepted by each of the four guards according to Table IV. The actual schedules are generated by searching for a path to a state where this input has been consumed and the *DCPred* actor is ready for the next control input. Then, we check if the

Guard	Nr. actions (State0)	Nr. actions (State1)
start	9	268
inter_ac	135	330
other	8	267
intra	136	331

TABLE V
THE LENGTH OF THE SCHEDULES GENERATED FOR THE EXAMPLE NETWORK. A SCHEDULE IS CHOSEN BASED ON WHICH GUARD EVALUATES TO TRUE AND THE CURRENT SCHEDULER STATE. THE NUMBERS INDICATE HOW MANY ACTIONS ARE FIRED BASED ON ONE GUARD EVALUATION.

found state is an already known state or if we must add this state to the scheduler. When a new state is required, schedules for each input type are also generated for this state. When each state has, in this case, four transitions connected to a known state, the scheduling is completed.

The scheduling of this network results in a scheduler with two states and eight transitions representing static schedules (see Table V). The reason for the second state is that in this particular implementation, the *zigzag* fills an internal buffer at the first block of a frame and flushes the buffer at a new frame, having different FSM states indicating whether the buffer is filled or not. This implies that two different schedules are needed for each block type. The scheduling variables named *count* each return to the initial value after each schedule, the scheduling variable *comp*, however, does not as it indicates which block of the macro block currently is processed. For this reason this scheduling decision needs to be taken at run-time, either by adding more states to the scheduler or, as in this case was possible by first applying the tools presented in [4], merging the actions depending on this variable and removing the variable from the scheduling state space.

A scheduler for the partition in this case study, can be described as in Table VI. This scheduler is somewhat simplified and indicates that we need to check that the control token is available and based on the guards from *DCPred*, choose the appropriate schedule. The schedules are sequences of actions that can fire without any further guard evaluations. Comparing this result to [5], we end up with a corresponding set of schedules for this network, and obviously similar results regarding speed-up; about 22% increase in frame rate for the texture coding part (acdc, idct) of the decoder. However, the scheduler is generated with evidence that the set of schedules completely describes the original program behavior and works for all possible input while in [5] this was verified manually by inspecting the code.

```
void decoder_acdc_scheduler() {

  if ( has_tokens(BTYPE, 1) )
    btype = peek(BTYPE)
  else return

  if (state == 0)
  {
    if (btype == grd_DCPred_start)
      dcPred_start();
      acPred_newvop();
      ...
      state = 0;
    else if (btype == grd_DCPred_inter_ac)
      dcPred_inter_ac();
      acPred_start();
      ...
      state = 1;
    else if ...
  }
  else if (state == 1)
  {
  ...
  }
}
```

TABLE VI
EXAMPLE PSEUDO CODE OF GENERATED SCHEDULER.

VI. CONCLUSIONS

In this paper we have shown how control tokens can be modelled to enable composition and scheduling of dynamic dataflow programs. Control token paths are analyzed in an abstract manner to deduce if there is a real data dependency or if a control token is used to distribute some information in the dataflow network, which, if the actors are composed, can be removed. As a result, many of the guards can be resolved at compile-time, resulting in less scheduling overhead at run-time. The actual speed-up a program gains, however, depends on how large partitions are composed and how well these fit on the target platform i.e. how well the schedules fit in the cache. The goal with the methods presented in this paper is to make the control tokens of a dataflow program a visible part of the model such that actors that can be efficiently composed are highlighted and can be scheduled using existing methods.

REFERENCES

[1] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[2] M. Mattavelli, I. Amer, and M. Raulet, "The reconfigurable video coding standard," *IEEE Signal Processing Magazine*, vol. 27, no. 3, pp. 157–167, 2010.

[3] J. Boutellier, M. Raulet, and O. Silvn, "Automatic hierarchical discovery of quasi-static schedules of rvc-cal dataflow programs," *Journal of Signal Processing Systems*, vol. 71, no. 1, pp. 35–40, 2013. [Online]. Available: http://dx.doi.org/10.1007/s11265-012-0676-4

[4] M. Wipliez and M. Raulet, "Classification of dataflow actors with satisfiability and abstract interpretation," *IJERTCS*, vol. 3, no. 1, pp. 49–69, 2012.

[5] J. Ersfolk, G. Roquier, F. Jokhio, J. Lilius, and M. Mattavelli, "Scheduling of dynamic dataflow programs with model checking," in *IEEE International Workshop on Signal Processing Systems (SiPS)*, 2011.

[6] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, "Cyclo-static data flow," in *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, vol. 5, May, pp. 3255–3258 vol.5.

[7] B. Bhattacharya and S. Bhattacharyya, "Parameterized dataflow modeling of dsp systems," in *Acoustics, Speech, and Signal Processing, 2000. ICASSP '00. Proceedings. 2000 IEEE International Conference on*, vol. 6, pp. 3362–3365 vol.6.

[8] R. Gu, J. Janneck, M. Raulet, and S. Bhattacharyya, "Exploiting statically schedulable regions in dataflow programs," *Journal of Signal Processing Systems*, vol. 63, pp. 129–142, 2011.

[9] J. Janneck, "A machine model for dataflow actors and its applications," in *Signals, Systems and Computers (ASILOMAR), 2011 Conference Record of the Forty Fifth Asilomar Conference on*, nov. 2011, pp. 756 –760.

[10] J. Ersfolk, G. Roquier, J. Lilius, and M. Mattavelli, "Scheduling of dynamic dataflow programs based on state space analysis," in *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, march 2012, pp. 1661 –1664.

Paper V

High-Performance Programs by Source-Level Merging of RVC-CAL Dataflow Actors

Jani Boutellier, Amanullah Ghazi, Olli Silvén, Johan Ersfolk

Originally published *2013 IEEE Workshop on Signal Processing Systems (SiPS)*. IEEE. October 2013, Taipei, Taiwan.

Turku Centre for Computer Science
TUCS Dissertations

1. **Marjo Lipponen**, On Primitive Solutions of the Post Correspondence Problem
2. **Timo Käkölä**, Dual Information Systems in Hyperknowledge Organizations
3. **Ville Leppänen**, Studies on the Realization of PRAM
4. **Cunsheng Ding**, Cryptographic Counter Generators
5. **Sami Viitanen**, Some New Global Optimization Algorithms
6. **Tapio Salakoski**, Representative Classification of Protein Structures
7. **Thomas Långbacka**, An Interactive Environment Supporting the Development of Formally Correct Programs
8. **Thomas Finne**, A Decision Support System for Improving Information Security
9. **Valeria Mihalache**, Cooperation, Communication, Control. Investigations on Grammar Systems.
10. **Marina Waldén**, Formal Reasoning About Distributed Algorithms
11. **Tero Laihonen**, Estimates on the Covering Radius When the Dual Distance is Known
12. **Lucian Ilie**, Decision Problems on Orders of Words
13. **Jukkapekka Hekanaho**, An Evolutionary Approach to Concept Learning
14. **Jouni Järvinen**, Knowledge Representation and Rough Sets
15. **Tomi Pasanen**, In-Place Algorithms for Sorting Problems
16. **Mika Johnsson**, Operational and Tactical Level Optimization in Printed Circuit Board Assembly
17. **Mats Aspnäs**, Multiprocessor Architecture and Programming: The Hathi-2 System
18. **Anna Mikhajlova**, Ensuring Correctness of Object and Component Systems
19. **Vesa Torvinen**, Construction and Evaluation of the Labour Game Method
20. **Jorma Boberg**, Cluster Analysis. A Mathematical Approach with Applications to Protein Structures
21. **Leonid Mikhajlov**, Software Reuse Mechanisms and Techniques: Safety Versus Flexibility
22. **Timo Kaukoranta**, Iterative and Hierarchical Methods for Codebook Generation in Vector Quantization
23. **Gábor Magyar**, On Solution Approaches for Some Industrially Motivated Combinatorial Optimization Problems
24. **Linas Laibinis**, Mechanised Formal Reasoning About Modular Programs
25. **Shuhua Liu**, Improving Executive Support in Strategic Scanning with Software Agent Systems
26. **Jaakko Järvi**, New Techniques in Generic Programming – C++ is more Intentional than Intended
27. **Jan-Christian Lehtinen**, Reproducing Kernel Splines in the Analysis of Medical Data
28. **Martin Büchi**, Safe Language Mechanisms for Modularization and Concurrency
29. **Elena Troubitsyna**, Stepwise Development of Dependable Systems
30. **Janne Näppi**, Computer-Assisted Diagnosis of Breast Calcifications
31. **Jianming Liang**, Dynamic Chest Images Analysis
32. **Tiberiu Seceleanu**, Systematic Design of Synchronous Digital Circuits
33. **Tero Aittokallio**, Characterization and Modelling of the Cardiorespiratory System in Sleep-Disordered Breathing
34. **Ivan Porres**, Modeling and Analyzing Software Behavior in UML
35. **Mauno Rönkkö**, Stepwise Development of Hybrid Systems
36. **Jouni Smed**, Production Planning in Printed Circuit Board Assembly
37. **Vesa Halava**, The Post Correspondence Problem for Market Morphisms
38. **Ion Petre**, Commutation Problems on Sets of Words and Formal Power Series
39. **Vladimir Kvassov**, Information Technology and the Productivity of Managerial Work
40. **Frank Tétard**, Managers, Fragmentation of Working Time, and Information Systems

41. **Jan Manuch**, Defect Theorems and Infinite Words
42. **Kalle Ranto**, Z_4-Goethals Codes, Decoding and Designs
43. **Arto Lepistö**, On Relations Between Local and Global Periodicity
44. **Mika Hirvensalo**, Studies on Boolean Functions Related to Quantum Computing
45. **Pentti Virtanen**, Measuring and Improving Component-Based Software Development
46. **Adekunle Okunoye**, Knowledge Management and Global Diversity – A Framework to Support Organisations in Developing Countries
47. **Antonina Kloptchenko**, Text Mining Based on the Prototype Matching Method
48. **Juha Kivijärvi**, Optimization Methods for Clustering
49. **Rimvydas Rukšėnas**, Formal Development of Concurrent Components
50. **Dirk Nowotka**, Periodicity and Unbordered Factors of Words
51. **Attila Gyenesei**, Discovering Frequent Fuzzy Patterns in Relations of Quantitative Attributes
52. **Petteri Kaitovaara**, Packaging of IT Services – Conceptual and Empirical Studies
53. **Petri Rosendahl**, Niho Type Cross-Correlation Functions and Related Equations
54. **Péter Majlender**, A Normative Approach to Possibility Theory and Soft Decision Support
55. **Seppo Virtanen**, A Framework for Rapid Design and Evaluation of Protocol Processors
56. **Tomas Eklund**, The Self-Organizing Map in Financial Benchmarking
57. **Mikael Collan**, Giga-Investments: Modelling the Valuation of Very Large Industrial Real Investments
58. **Dag Björklund**, A Kernel Language for Unified Code Synthesis
59. **Shengnan Han**, Understanding User Adoption of Mobile Technology: Focusing on Physicians in Finland
60. **Irina Georgescu**, Rational Choice and Revealed Preference: A Fuzzy Approach
61. **Ping Yan**, Limit Cycles for Generalized Liénard-Type and Lotka-Volterra Systems
62. **Joonas Lehtinen**, Coding of Wavelet-Transformed Images
63. **Tommi Meskanen,** On the NTRU Cryptosystem
64. **Saeed Salehi**, Varieties of Tree Languages
65. **Jukka Arvo**, Efficient Algorithms for Hardware-Accelerated Shadow Computation
66. **Mika Hirvikorpi**, On the Tactical Level Production Planning in Flexible Manufacturing Systems
67. **Adrian Costea**, Computational Intelligence Methods for Quantitative Data Mining
68. **Cristina Seceleanu**, A Methodology for Constructing Correct Reactive Systems
69. **Luigia Petre**, Modeling with Action Systems
70. **Lu Yan**, Systematic Design of Ubiquitous Systems
71. **Mehran Gomari**, On the Generalization Ability of Bayesian Neural Networks
72. **Ville Harkke**, Knowledge Freedom for Medical Professionals – An Evaluation Study of a Mobile Information System for Physicians in Finland
73. **Marius Cosmin Codrea**, Pattern Analysis of Chlorophyll Fluorescence Signals
74. **Aiying Rong**, Cogeneration Planning Under the Deregulated Power Market and Emissions Trading Scheme
75. **Chihab BenMoussa**, Supporting the Sales Force through Mobile Information and Communication Technologies: Focusing on the Pharmaceutical Sales Force
76. **Jussi Salmi**, Improving Data Analysis in Proteomics
77. **Orieta Celiku**, Mechanized Reasoning for Dually-Nondeterministic and Probabilistic Programs
78. **Kaj-Mikael Björk**, Supply Chain Efficiency with Some Forest Industry Improvements
79. **Viorel Preoteasa**, Program Variables – The Core of Mechanical Reasoning about Imperative Programs
80. **Jonne Poikonen**, Absolute Value Extraction and Order Statistic Filtering for a Mixed-Mode Array Image Processor
81. **Luka Milovanov**, Agile Software Development in an Academic Environment
82. **Francisco Augusto Alcaraz Garcia**, Real Options, Default Risk and Soft Applications
83. **Kai K. Kimppa**, Problems with the Justification of Intellectual Property Rights in Relation to Software and Other Digitally Distributable Media
84. **Dragoş Truşcan**, Model Driven Development of Programmable Architectures
85. **Eugen Czeizler**, The Inverse Neighborhood Problem and Applications of Welch Sets in Automata Theory

86. **Sanna Ranto**, Identifying and Locating-Dominating Codes in Binary Hamming Spaces
87. **Tuomas Hakkarainen**, On the Computation of the Class Numbers of Real Abelian Fields
88. **Elena Czeizler**, Intricacies of Word Equations
89. **Marcus Alanen**, A Metamodeling Framework for Software Engineering
90. **Filip Ginter**, Towards Information Extraction in the Biomedical Domain: Methods and Resources
91. **Jarkko Paavola**, Signature Ensembles and Receiver Structures for Oversaturated Synchronous DS-CDMA Systems
92. **Arho Virkki**, The Human Respiratory System: Modelling, Analysis and Control
93. **Olli Luoma**, Efficient Methods for Storing and Querying XML Data with Relational Databases
94. **Dubravka Ilić**, Formal Reasoning about Dependability in Model-Driven Development
95. **Kim Solin**, Abstract Algebra of Program Refinement
96. **Tomi Westerlund**, Time Aware Modelling and Analysis of Systems-on-Chip
97. **Kalle Saari**, On the Frequency and Periodicity of Infinite Words
98. **Tomi Kärki**, Similarity Relations on Words: Relational Codes and Periods
99. **Markus M. Mäkelä**, Essays on Software Product Development: A Strategic Management Viewpoint
100. **Roope Vehkalahti**, Class Field Theoretic Methods in the Design of Lattice Signal Constellations
101. **Anne-Maria Ernvall-Hytönen**, On Short Exponential Sums Involving Fourier Coefficients of Holomorphic Cusp Forms
102. **Chang Li**, Parallelism and Complexity in Gene Assembly
103. **Tapio Pahikkala**, New Kernel Functions and Learning Methods for Text and Data Mining
104. **Denis Shestakov**, Search Interfaces on the Web: Querying and Characterizing
105. **Sampo Pyysalo**, A Dependency Parsing Approach to Biomedical Text Mining
106. **Anna Sell**, Mobile Digital Calendars in Knowledge Work
107. **Dorina Marghescu**, Evaluating Multidimensional Visualization Techniques in Data Mining Tasks
108. **Tero Säntti**, A Co-Processor Approach for Efficient Java Execution in Embedded Systems
109. **Kari Salonen**, Setup Optimization in High-Mix Surface Mount PCB Assembly
110. **Pontus Boström**, Formal Design and Verification of Systems Using Domain-Specific Languages
111. **Camilla J. Hollanti**, Order-Theoretic Mehtods for Space-Time Coding: Symmetric and Asymmetric Designs
112. **Heidi Himmanen**, On Transmission System Design for Wireless Broadcasting
113. **Sébastien Lafond**, Simulation of Embedded Systems for Energy Consumption Estimation
114. **Evgeni Tsivtsivadze**, Learning Preferences with Kernel-Based Methods
115. **Petri Salmela**, On Commutation and Conjugacy of Rational Languages and the Fixed Point Method
116. **Siamak Taati**, Conservation Laws in Cellular Automata
117. **Vladimir Rogojin**, Gene Assembly in Stichotrichous Ciliates: Elementary Operations, Parallelism and Computation
118. **Alexey Dudkov**, Chip and Signature Interleaving in DS CDMA Systems
119. **Janne Savela**, Role of Selected Spectral Attributes in the Perception of Synthetic Vowels
120. **Kristian Nybom**, Low-Density Parity-Check Codes for Wireless Datacast Networks
121. **Johanna Tuominen**, Formal Power Analysis of Systems-on-Chip
122. **Teijo Lehtonen**, On Fault Tolerance Methods for Networks-on-Chip
123. **Eeva Suvitie**, On Inner Products Involving Holomorphic Cusp Forms and Maass Forms
124. **Linda Mannila**, Teaching Mathematics and Programming – New Approaches with Empirical Evaluation
125. **Hanna Suominen**, Machine Learning and Clinical Text: Supporting Health Information Flow
126. **Tuomo Saarni**, Segmental Durations of Speech
127. **Johannes Eriksson**, Tool-Supported Invariant-Based Programming

128. **Tero Jokela**, Design and Analysis of Forward Error Control Coding and Signaling for Guaranteeing QoS in Wireless Broadcast Systems
129. **Ville Lukkarila**, On Undecidable Dynamical Properties of Reversible One-Dimensional Cellular Automata
130. **Qaisar Ahmad Malik**, Combining Model-Based Testing and Stepwise Formal Development
131. **Mikko-Jussi Laakso**, Promoting Programming Learning: Engagement, Automatic Assessment with Immediate Feedback in Visualizations
132. **Riikka Vuokko**, A Practice Perspective on Organizational Implementation of Information Technology
133. **Jeanette Heidenberg**, Towards Increased Productivity and Quality in Software Development Using Agile, Lean and Collaborative Approaches
134. **Yong Liu**, Solving the Puzzle of Mobile Learning Adoption
135. **Stina Ojala**, Towards an Integrative Information Society: Studies on Individuality in Speech and Sign
136. **Matteo Brunelli**, Some Advances in Mathematical Models for Preference Relations
137. **Ville Junnila**, On Identifying and Locating-Dominating Codes
138. **Andrzej Mizera**, Methods for Construction and Analysis of Computational Models in Systems Biology. Applications to the Modelling of the Heat Shock Response and the Self-Assembly of Intermediate Filaments.
139. **Csaba Ráduly-Baka**, Algorithmic Solutions for Combinatorial Problems in Resource Management of Manufacturing Environments
140. **Jari Kyngäs**, Solving Challenging Real-World Scheduling Problems
141. **Arho Suominen**, Notes on Emerging Technologies
142. **József Mezei**, A Quantitative View on Fuzzy Numbers
143. **Marta Olszewska**, On the Impact of Rigorous Approaches on the Quality of Development
144. **Antti Airola**, Kernel-Based Ranking: Methods for Learning and Performace Estimation
145. **Aleksi Saarela**, Word Equations and Related Topics: Independence, Decidability and Characterizations
146. **Lasse Bergroth**, Kahden merkkijonon pisimmän yhteisen alijonon ongelma ja sen ratkaiseminen
147. **Thomas Canhao Xu**, Hardware/Software Co-Design for Multicore Architectures
148. **Tuomas Mäkilä**, Software Development Process Modeling – Developers Perspective to Contemporary Modeling Techniques
149. **Shahrokh Nikou**, Opening the Black-Box of IT Artifacts: Looking into Mobile Service Characteristics and Individual Perception
150. **Alessandro Buoni**, Fraud Detection in the Banking Sector: A Multi-Agent Approach
151. **Mats Neovius**, Trustworthy Context Dependency in Ubiquitous Systems
152. **Fredrik Degerlund**, Scheduling of Guarded Command Based Models
153. **Amir-Mohammad Rahmani-Sane**, Exploration and Design of Power-Efficient Networked Many-Core Systems
154. **Ville Rantala**, On Dynamic Monitoring Methods for Networks-on-Chip
155. **Mikko Pelto**, On Identifying and Locating-Dominating Codes in the Infinite King Grid
156. **Anton Tarasyuk**, Formal Development and Quantitative Verification of Dependable Systems
157. **Muhammad Mohsin Saleemi**, Towards Combining Interactive Mobile TV and Smart Spaces: Architectures, Tools and Application Development
158. **Tommi J. M. Lehtinen**, Numbers and Languages
159. **Peter Sarlin**, Mapping Financial Stability
160. **Alexander Wei Yin**, On Energy Efficient Computing Platforms
161. **Mikołaj Olszewski**, Scaling Up Stepwise Feature Introduction to Construction of Large Software Systems
162. **Maryam Kamali**, Reusable Formal Architectures for Networked Systems
163. **Zhiyuan Yao**, Visual Customer Segmentation and Behavior Analysis – A SOM-Based Approach
164. **Timo Jolivet**, Combinatorics of Pisot Substitutions
165. **Rajeev Kumar Kanth**, Analysis and Life Cycle Assessment of Printed Antennas for Sustainable Wireless Systems
166. **Khalid Latif**, Design Space Exploration for MPSoC Architectures

167. **Bo Yang**, Towards Optimal Application Mapping for Energy-Efficient Many-Core Platforms
168. **Ali Hanzala Khan**, Consistency of UML Based Designs Using Ontology Reasoners
169. **Sonja Leskinen**, m-Equine: IS Support for the Horse Industry
170. **Fareed Ahmed Jokhio**, Video Transcoding in a Distributed Cloud Computing Environment
171. **Moazzam Fareed Niazi**, A Model-Based Development and Verification Framework for Distributed System-on-Chip Architecture
172. **Mari Huova**, Combinatorics on Words: New Aspects on Avoidability, Defect Effect, Equations and Palindromes
173. **Ville Timonen**, Scalable Algorithms for Height Field Illumination
174. **Henri Korvela**, Virtual Communities – A Virtual Treasure Trove for End-User Developers
175. **Kameswar Rao Vaddina**, Thermal-Aware Networked Many-Core Systems
176. **Janne Lahtiranta**, New and Emerging Challenges of the ICT-Mediated Health and Well-Being Services
177. **Irum Rauf**, Design and Validation of Stateful Composite RESTful Web Services
178. **Jari Björne**, Biomedical Event Extraction with Machine Learning
179. **Katri Haverinen**, Natural Language Processing Resources for Finnish: Corpus Development in the General and Clinical Domains
180. **Ville Salo**, Subshifts with Simple Cellular Automata
181. **Johan Ersfolk**, Scheduling Dynamic Dataflow Graphs

Turku Centre *for* Computer Science

Joukahaisenkatu 3-5 B, 20520 Turku, Finland | www. tucs.fi

University of Turku
Faculty of Mathematics and Natural Sciences
- Department of Information Technology
- Department of Mathematics and Statistics

Turku School of Economics
- Institute of Information Systems Science

Åbo Akademi University
Division for Natural Sciences and Technology
- Department of Information Technologies

ISBN 978-952-12-3091-2
ISSN 1239-1883

Johan Ersfolk

Scheduling Dynamic Dataflow with Model Checking

Johan Ersfolk

Scheduling Dynamic Dataflow Graphs with Model Checking