AUTOMATED DISTRIBUTED SIMULATION IN PTOLEMY II

by

DANIEL LÁZARO CUADRADO

EMBEDDED SYSTEMS GROUP
Center for TeleInfratructure (CTIF)
Aalborg University (AaU)
Fredrik Bajers Vej 7, A3
DK-9220 Aalborg
Denmark

Supervisors

Prof. Anders P. Ravn, Aalborg University, Denmark
Assoc. Prof. Peter Koch, Aalborg University, Denmark

Opponents

Prof. Brian Vinter, University of Copenhagen, Denmark
Assoc. Prof. Kallol Bagchi, University of Texas at El Paso, USA
Assoc. Prof. Josva Kleist, Aalborg University, Denmark (chairman)

ISSN 0908-1224
ISBN 87-92078-08-7

A mis padres, que valen más que cien maestros.

ABSTRACT

Despite the well known advantages of distributed processing for intensive computations like simulation, frameworks often fail to exploit them. A distributed simulation is harder to develop than a sequential one, because it is necessary to interface and map activities to processors and handle the ensuing communication and synchronization problems. Very often the designer has to explicitly specify information about distribution for the framework to exploit parallelism. This dissertation presents Automated Distributed Simulation (ADS), which allows designers to forget about distribution concerns while benefiting from the advantages. This study shows abstractions that help automate the distribution of a simulation by taking maximum advantage of inherent parallelism. ADS relies on an actor formalism to provide encapsulation of component behavior and reusability. Models of computation govern component interactions by defining execution and communication mechanisms and the notion of time; thus providing semantics. Different models of computation useful for embedded systems are surveyed to discuss possibilities for distribution. Synchronous Dataflow allows for static scheduling, therefore eliminating runtime overheads and having a higher potential for efficient parallelization. Moreover it is a popular formalism suited for a large number of embedded applications, thus is chosen for the initial implementation of ADS. We present a novel execution mechanism that produces optimal periodic admissible parallel schedules allowing the dispatching mechanism to apply pipelining techniques. The implementation is described with major emphasis on distribution issues as interfacing and mapping of activities, communication and synchronization. We have chosen Ptolemy II as the implementation framework since it provides the abstractions required to achieve ADS as open source. The Ptolemy project studies heterogeneous modeling, simulation and design of concurrent real-time, embedded systems; Ptolemy II is the current software incarnation. Experiments to explore the gain for varying values of relevant parameters such as block size, number of blocks, number of iterations and topology have been designed and run. Analytical expressions for the expected results are derived and compared with the empirical results to arrive at relative overheads. The experiments show significant gains over the original implementation. The topology of a dataflow model plays an important role. However, serialization can be mitigated by using pipelined execution. Time saved in parallelization plus linear speedup in makespans provided by ADS can help tackle hitherto infeasible simulations.

The implementation result of this work is publicly available as a new feature of the latest public release of Ptolemy II. Since it is open source, it constitutes a major mean of dissemination of the ideas presented here, as well as documentation for this work.

DANSK RESUMÉ[1]

Til trods for de velkendte fordele ved at udføre intensive beregninger for eksempel simuleringer med flere distribuerede processorer udnyttes det ikke meget. En distribueret simulering er sværere at udvikle end en sekventiel, da det er nødvendigt at opdele i aktiviteter, binde dem til processorer og løse de medfølgende kommunikations- og synkroniseringsopgaver. Meget ofte må designeren specificere ekstra information om bindingen, for at simuleringen kan udnytte den parallelle udførelse. Denne afhandling præsenterer "Automated Distributed Simulation" (ADS), som lader designeren glemme alt om distribueringsopgaven og stadigvæk få fordelene. ADS automatiserer distributionen af en simulering ved at udnytte den iboende parallelisme i aktorbegrebet, idet en aktør indkapsler komponentadfærd og fremmer genbrug. En aktør realiserer en beregningsmodel, der styrer komponent-interaktion ved at definere udførelses- og kommunikationsmekanismer samt tidsbegrebet; dermed fastlægger aktøren semantikken. Forskellige beregningsmodeller, som er brugbare for indlejrede systemer undersøges for at klarlægge mulighederne for distribution. "Synchronous Data Flow" (SDF) er en velegnet beregningsmodel. SDF har statisk afvikling af beregningsforløb, og det gives et højere potentiale for statisk distribution. Desuden er det en populær model, hvorfor den er valgt til prototype for implementering af ADS. I SDF indfører vi en ny afviklingsplan, som producerer optimale periodiske parallelle planer, som tillader kommunikationsmekanismerne at anvende buffere. Implementeringen beskrives med stor vægt på distributionsproblemer såsom opdeling og binding af aktiviteter, kommunikation og synkronisering. Ptolemy II er valgt som basis for implementeringen, da det leverer de abstraktioner, der kræves for at opnå ADS. Ptolemy-projektet undersøger heterogen modellering, simulering og design af tidstro indlejrede systemer, og Ptolemy II er den nuværende realisering. Afhandlingen beskriver en række forsøg, der udforsker betydningen af relevante parametre for SDF modeller såsom blokstørrelse, antal af blokke, antal af iterationer og topologi. Analytiske udtryk for de forventede resultater uden kommunikationsomkostninger udledes og sammenlignes med de eksperimentelle resultater. Forsøgene viser, at de forventede forbedringer opnås med en minimal og lineært voksende ekstraomkostning. SDF-modellens topologi kan forhindre effektiv parallelisering, men det kan modvirkes ved at bruge buffere ved udførelsen.

ADS implementeringen beskrevet i dette værk er offentligt tilgængeligt som en del af den seneste offentlige version af Ptolemy II. Da det er open source, udgør det et stort medium for offentliggørelse af de præsenterede ideer samt dokumentation for resultaterne i denne afhandling.

[1] Translation by: Eva Hansen, Rikke Svendsen, Martin Lindberg and Anders P. Ravn

Every noble work is bound to face problems and obstacles. It is important to check your goal and motivation thoroughly. One should be very truthful, honest, and reasonable. One's actions should be good for others, and for oneself as well. Once a positive goal is chosen, you should decide to pursue it all the way to the end. Even if it is not realized, at least there will be no regret.

Dalai Lama

ACKNOWLEDGEMENTS

First of all I would like to express my gratitude to Associate Professor Peter Koch for introducing me to the research world, giving me the opportunity to perform this work and gathering enough belief at times of slow progress. Professor Anders P. Ravn has greatly contributed to this work with his time, patience, constructive advice and willingness to share his insight and wisdom. His tireless support and effort has made me a better scholar and writer. Ole Olsen was a great source of inspiration when dealing with teaching. His educational values and involvement with students are utterly missed since his retirement. My colleagues at the Embedded Systems Group have played an important role in creating a warm atmosphere in which to work. I would like to thank the Ptolemy Group in general and E. A. Lee in particular for the invitation to visit the Ptolemy Group in summer 2003. They made me feel part of the group and at home, both in the literal and figurative senses. This dissertation was greatly inspired by and partly developed during this stay.

After few years in the ever changing international environment at Aalborg University and several trips, I have had the chance to meet a great variety of people. Most of them have inspired me, contributing to different extents to make me who I am, enriching me as a person. Having left all behind, and coming from another country and culture, they have meant the world to the life of this Ph. D. candidate by becoming my "family" out here. They have provided love, friendship, laughs, company, support, encouragement and understanding among many other very valuable "assets", to overcome the loneliness and isolation that comes with Ph.D. studies. I cherish every single second we have shared and I will do my best to keep these bonds alive despite the distance. The fact that mentioning all the names and stories would take several volumes makes me feel so lucky, and anyways, we both know who you are, don't we? Nevertheless, I cannot forget Tamán González. He was assigned my "buddy" when I first arrived to Denmark. I know you would have rather had a girl to look after instead of me! … but I have to thank you for becoming my best friend, being so supportive and understanding all these years.

Pals back home have always welcomed me when I returned despite the distance. Especially my family has always been there making me feel missed. I have missed them all so much and I am undoubtedly grateful for all the unconditional love and positive energy.

Rikke Svendsen has been standing very patiently by my side during the writing phase. Her love and affection have been incommensurable. Tú eres mi más y mi menos.

With such fantastic supporters, one knows one has won… even before the match has started.

TABLE OF CONTENTS

ABBREVIATIONS

µC	**MICRO CONTROLLER**
3G	**THIRD GENERATION**
4G	**FOUTH GENERATION**
ADC	**ANALOG TO DIGITAL CONVERTER**
ADS	**AUTOMATED DISTRIBUTED SIMULATION**
AFS	**ANDREW FILESYSTEM**
APEG	**ACYCLIC PRECEDENCE EXPANSION GRAPH**
API	**APPLICATION PROGRAMMING INTERFACE**
ASIC	**APPLICATION SPECIFIC INTEGRATED CIRCUIT**
ASM	**ABSTRACT STATE MACHINE**
ASML	**ABSTRACT STATE MACHINE LANGUAGE**
ATM	**AUTOMATIC TELLER MACHINE**
BDF	**BOOLEAN DATAFLOW**
CCR	**COMMUNICATION TO COMPUTATION RATIO**
CORBA	**COMMON OBJECT REQUEST BROKER ARCHITECTURE**
CSDF	**CYCLO-STATIC DATAFLOW**
CSP	**COMMUNICATING SEQUENTIAL PROCESSES**
CVS	**CONCURRENT VERSIONS SYSTEM**
DAG	**DIRECTED ACYCLIC GRAPH**
DCOM	**DISTRIBUTED COMPONENT OBJECT MODEL**
DDE	**DISTRIBUTED DISCRETE EVENTS**
DDF	**DYNAMIC DATAFLOW**
DE	**DISCRETE EVENTS**
DSP	**DIGITAL SIGNAL PROCESSOR**
EDA	**ELECTRONIC DESIGN AUTOMATION**
ES	**EMBEDDED SYSTEM**
FSM	**FINITE STATE MACHINE**
GHZ	**GIGA HERTZ**
HDF	**HETEROCHRONOUS DATAFLOW**
HPC	**HIGH PERFORMANCE COMPUTING**
IP	**INTELLECTUAL PROPERTY**
IPC	**INTER-PROCESSOR COMMUNICATION**
JINI	**JINI IS NOT INITIALS**
JVM	**JAVA VIRTUAL MACHINE**
JXTA	**JUXTAPOSE**

MOC	**MODEL OF COMPUTATION**
MOML	**MODELING MARKUP LANGUAGE**
MPEG	**MOVING PICTURE EXPERTS GROUP**
MPI	**MESSAGE PASSING INTERFACE**
MRIP	**MULTIPLE REPLICATION IN PARALLEL**
ODE	**ORDINARY DIFFERENTIAL EQUATION**
OS	**OPERATIVE SYSTEM**
P2P	**PEER TO PEER**
PASS	**PERIODIC ADMISSIBLE SEQUENTIAL SCHEDULE**
PAPS	**PEDIODIC ADMISSIBLE PARALLEL SCHEDULE**
PDA	**PERSONAL DIGITAL ASSISTANT**
PE	**PROCESSING ELEMENT**
PN	**PROCESS NETWORKS**
RMI	**REMOTE METHOD INVOCATION**
RPC	**REMOTE PROCEDURE CALL**
SDF	**SYNCHRONOUS DATAFLOW**
SOAP	**SIMPLE OBJECT ACCESS PROTOCOL**
SRIP	**SINGLE REPLICATION IN PARALLEL**
TTL	**TIME TO LIVE**
TTM	**TIME TO MARKET**
UDP	**USER DATAGRAM PROTOCOL**
UML	**UNIFIED MODELING LANGUAGE**
VHDL	**VHSIC HARDWARE DESCRIPTION LANGUAGE**
VHSIC	**VERY HIGH SPEED INTEGRATED CIRCUIT**
XML	**EXTENSIBLE MARKUP LANGUAGE**

CHAPTER 1

INTRODUCTION

All of the biggest technological inventions created by man - the airplane, the automobile, the computer - says little about his intelligence, but speaks volumes about his laziness.
Mark Kennedy

Nowadays, there is an increasing amount of convenient gadgets and devices in our daily lives. Most of them contain embedded software and circuits. Personal Digital Assistants (PDAs), Automatic Teller Machines (ATMs), digital watches, microwave ovens, MP3 portable players, videogame consoles, cars and mobile phones are a few examples of our daily use and interaction with embedded systems.

1.1 EMBEDDED SYSTEMS

An *embedded system* (ES) is an application area which has gained interest in the scientific community in the past few years. They have evolved from simple electronic systems to complex distributed systems combining hardware and software. As stated in [1], there is no clear-cut definition for embedded systems, and the definition continues to blur as the application areas expand. We will view them as engineering artifacts that are generally an integral part of a larger heterogeneous system. It is often the case that the computer is encapsulated by the device it controls. Embedded systems are unlike general-purpose computers devoted to specific tasks. Some are bounded by very specific requirements like real-time performance constraints for reasons as for example safety and usability. Others may have low or no performance requirements, often used as a base for optimization and cost reduction.

1.2 INTERDISCIPLINARITY AND HETEROGENEITY

Embedded Systems is an *interdisciplinary* area that combines theories and techniques from many different fields, for instance electronics, hardware design, computer science and telecommunications [2]. Since they appear in a wide variety of industrial sectors, they require different skills. These are mainly design skills like

software architecture design and hardware design and fault tolerant design plus others like safety techniques, verification and testing. From the electronics area we can find components built for specific purposes as for example, Analog-to-Digital converters (ADC). Furthermore, the hardware parts combine different technologies like Application Specific Integrated Circuits (ASICs), Intellectual Properties (IPs), Digital Signal Processors (DSPs) [3] and Micro-controllers (μCs). Different languages exist for hardware design like Verilog, VHDL and SystemC [4]. From the software side we can find different computational models for reactive systems as for instance: Dataflow Networks, Automata, Discrete Event Systems and Process Algebras. There exist different languages to develop software for embedded systems as Assembly Languages, C, C++ or Java. Different Operating Systems (OSs) usually glue everything together. Many of the functionalities traditionally implemented on hardware are now being migrated to software due for example to cost reduction. Modern embedded systems are highly dependent on software.

1.3 SOFTWARE

Computer software (or simply *software*) refers to one or more computer programs and data held in the storage of a computer for some purpose. Software performs the function of the program it implements, either by directly providing instructions to the computer hardware or by serving as input to another piece of software. The term *software* was first used in this sense by John W. Tukey in 1957. In computer science and software engineering, software is all the information processing provided by the computer programs and data. Software is often contrasted with hardware, which is the physical substrate on which software exists.

1.4 EMBEDDED SOFTWARE

We will refer to the software that resides inside an embedded system as *embedded software* [5]. It is software whose principal role is not the transformation of data, but rather the interaction with the physical world. We can distinguish two types of interactions, reacting to the physical environment and executing on a physical platform. Therefore it must acquire some properties from the physical world as *time*, *power consumption* and *non-termination*. It executes on machines that are not first and foremost computers. Embedded software is sometimes called *firmware* because it is stored on ROM (Read Only Memory) or Flash memory chips rather than a disk drive. Often, it is run on limited hardware resources and/or without user interfaces.

1.5 GROWING PERVASIVENESS, COMPLEXITY AND COST

Within the Information and Telecommunication Technology Revolution Era, the number and complexity of embedded systems has increased dramatically. The third generation (3G) mobile phones already available in the market are good examples of state-of-the-art embedded systems. The trend is that more systems and objects will

contain computer-controlled components as in automobiles, trains, planes, military and consumer electronics. Furthermore, the new up-coming forth generation (4G) mobile phones envision a yet more demanding picture in timing properties. The signal processing algorithms will be much more complicated to develop because for example, they will implement *cognitive radio* concepts [6]. Higher bandwidth capabilities (100 Mbps wide coverage and 1 Gbps local area [7]) will increase the number of different applications that will partly be time critical such as video conference. On top of that, resource consumption will be a highly competitive parameter. This gives an idea of how complex, heterogeneous and yet constrained future products will be. Such systems will play a key role in the future mass market for high-quality wireless multimedia communications. Moreover, car manufacturers realize as well the importance of electronics in their business: more than 90% of innovation in a car will be in electronics (Daimler-Chrysler). More than 30% of the cost of manufacturing a car resides in the electronic components (BMW). In the cost of developing a new airplane (of the order of several billions of Euros), about half is related to embedded software and electronic subsystems. All those systems will contain significant amounts of complex embedded software that will have to be developed within constrained time-to-market (TTM) deadlines.

1.6 CHALLENGES IN EMBEDDED SYSTEMS DESIGN

Traditionally, embedded systems were either simple, or the composition of almost non-interacting components, so that a single person or a small group of people was able to complete the design. The trend is that the number and complexity of functionalities is increasing drastically. There is a need for a revolutionary approach to *embedded systems design*. Systems design is the process of deriving from requirements a *model* from which a system can be generated more or less automatically. A *model* is an abstract representation of a system.

1.6.1 ABSTRACTIONS

Among the myriad of challenges embedded system design poses, there seems to be a general consensus that the key to overcome them is *abstractions*. To invent or apply *abstractions* that yield more understandable programs [8]. Better abstractions that do not abstract away from properties that are most important in an embedded system [9]. Raise the levels of abstraction of designs [1]. The key problem becomes identifying the appropriate abstractions for representing the design [10]. The purpose for an abstraction is to hide the details of the implementation below and provide a platform for design from above. Furthermore, existing Computer Science paradigms have to be enriched and combined with methods found in other areas such as electronic engineering and embrace interdisciplinarity. Untimed and sequential programming abstractions are insufficient, particularly in embedded systems which are intrinsically timed and distributed. There is a need for building a new scientific foundation that integrates computation and physicality.

1.6.2 TOOLS

Therefore, there is a need for *tools* that implement new concepts and such abstractions. Tools must at the same time contain software *frameworks*. We consider frameworks as in the broad definition in [10]. Generally there is a trade-off between the benefits obtained by using frameworks and the time spent on learning them. Software frameworks are designed with the intent of facilitating software development, by allowing designers and programmers to spend more time on meeting software requirements rather than dealing with the more tedious low level details of providing a working system. Frameworks impose constraints on *components* interaction, thus defining a *model of computation* (MoC) and a set of benefits derive from those constraints. For example, limiting the choices during development, so it increases productivity, specifically in big complex systems. Frameworks help develop and glue together the different components of a design. If we do not have a breakthrough in design methodology and tools, the inefficiency of the embedded software development process will prevent novel technology to enter the market in time.

1.7 SIMULATION

A widely used approach in current embedded systems design practice is *simulation* [1]. A *computer simulation* is a computation that models the behavior of a system over time. Simulation relies on models, which are abstract simplified imitations of systems. Some models represent *static* properties of the system, such as its functionality or physical dimensions. A model can also be constructive, in which case it defines a computational procedure that mimics the behavior of the system. Constructive models are executable and are often used to describe behavior of a system in response to stimulus from the outside. The models used in a simulation have traditionally been mathematical. When analysis is not possible, computer simulations are used as a substitute. Computer based *dynamic* simulations are often performed for embedded systems, which can be defined as "An imitation (on a computer) of a system at it progresses through time." [12]. Computer simulations are widely used today to analyze and foresee aspects of the simulated system's behavior without actually constructing it. They are normally applied in verification and validation. When a complete enumeration of all reachable states of the model is impossible, computer simulations attempt to generate a sample of representative scenarios which require running the simulations several times. This, together with increasing size and complexity of models, results in high demands of computation hence execution time.

Single processor solutions cannot tackle such demands. Despite the optimistic formulations of Moore's Law (see Figure 1), and the exponential increase in frequency of operation of microprocessors what matters when increasing computation is the amount of work (instructions and their complexity) executed per unit time. Newer processors are actually being made at lower clock speeds, with focus on larger

caches and multiple computing cores since higher clock speeds correspond to exponential increases in temperature (it is almost impossible to produce processors that run reliably at speeds higher than 4.3 GHz). Furthermore, Moore himself stated that the law may not hold for too long, since transistors may reach fundamental atomic limits.

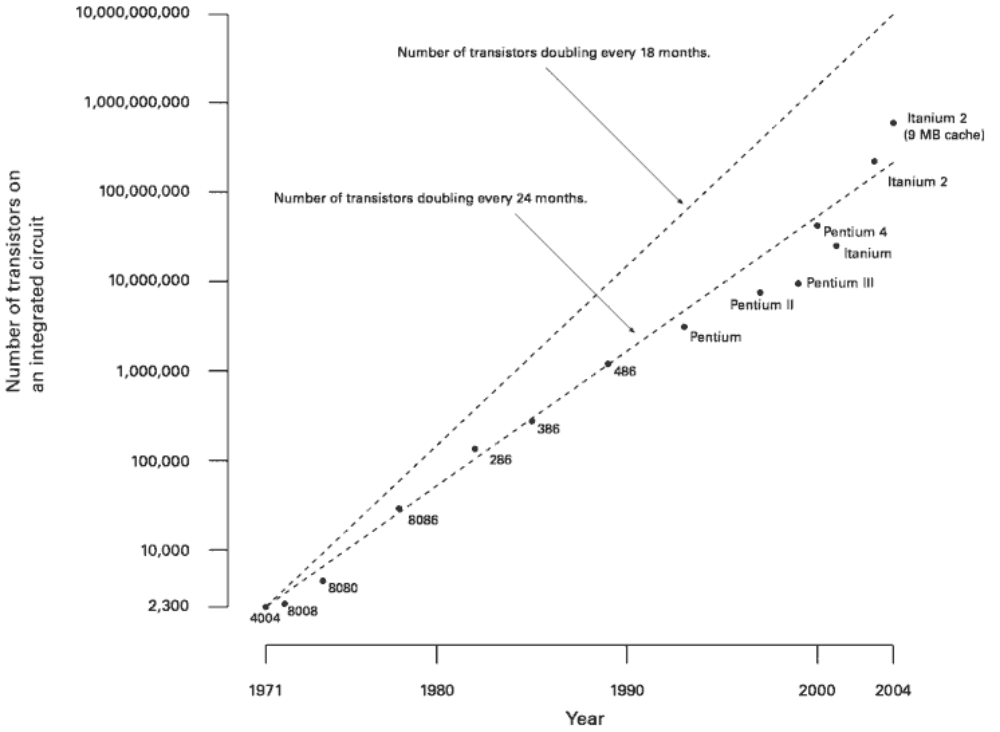


FIGURE 1: MOORE'S LAW

While a system of n (parallel) processors is less efficient than an n-times faster processor, the parallel system is often cheaper to build. *Distributed computation* is a cost efficient alternative widely used to tackle high computation demands.

1.8 MOTIVATION

Distributed computation enables a simulation program to execute on a computing system containing multiple processors. It is widely used for speeding up simulation, enabling the analysis of complex systems by concurrently exploiting the aggregate computation power and memory space of multiple computers communicating over a network. In practice, developing a distributed simulation is harder than a sequential because it is necessary to deal with the issues of distributed computation, interfacing and mapping activities to processors and handling the ensuing communication and synchronization problems [11]. On top of that, very often the designer has to explicitly specify extra information concerning distribution and deployment concerns for the framework to make an effort to exploit parallelism. The goal of automatic parallelization (distribution) is to relieve programmers from the tedious and error-prone manual parallelization process. Classical sequential programs require dependency analysis in order to make an effort to parallelize. Computing

dependencies with loops is a significant and nontrivial problem. Most of the execution time of a program takes place inside some form of loop. Though highly improved, full automatic parallelization of sequential programs remains a great challenge due to the complex program analysis required. Therefore, most widely used simulation environments fail to exploit the advantages of distributed simulation in their implementations. Abstractions such as the actor model and computational models used embedded systems modeling and design, provide the framework and information necessary for automating distributed simulations.

1.9 OBJECTIVE

In this thesis we propose Automated Distributed Simulation (ADS), which allows a designer to ignore distribution concerns while greatly benefiting from the advantages. ADS describes an architecture that deals with mapping of processes, communication and synchronization in a manner transparent to the user.

ADS mainly relies and builds on top of two abstractions:

- The actor formalism:
- A computational model (dataflow):

The actor formalism [13] defines independent concurrent processes, providing autonomous entities that are portable. Models of computation (MoC) plus a topology define the way in which actors communicate. This constitutes sufficient information to automate deployment for a simulation implemented with this paradigm. An actor-based framework providing MoCs is Ptolemy II [14]. Ptolemy II is a Java-based design environment that supports the construction and execution of hierarchical, reconfigurable models using actors for system level design. The actor formalism provides strong encapsulation separating the behavior of a component from the interaction with other components. Thus constraining input data range. Connections between actor ports represent communication channels, therefore making data dependencies explicit in the models. Models of computation define the semantics of composition, including the communication style. This separation of concerns [15] makes models easier to understand, write, reuse and modify.

Simulations in the Ptolemy II [16] tool are performed on one machine, either sequentially or threaded, but sharing the same CPU. There are also memory limitations given by the locally installed memory and the JVM (Java Virtual Machine). As with all simulation tools, users want the results quickly. And this without having to modify the models or any other extra complexities involved. The main contribution of this thesis is to implement ADS inside of Ptolemy II. Ptolemy contains many computational models or *domains*. They are implemented within a common framework, but have their own scheduling disciplines which reflect their semantics.

It is common practice to architect and realize concurrent mechanisms at a low level, as for example with monitors and critical sections. Dataflow-oriented models expose large amounts of concurrency at a system level. Automating the effective utilization of such concurrency through distributed computation is the main goal of this work. Of the provided MoCs, we have chosen the Synchronous Dataflow (SDF) domain for implementation, because it allows for static scheduling, and it is well suited for modeling a large number of applications. We believe that our approach can be applied to other MoCs.

Furthermore, we show that the overhead introduced by distribution is acceptable considering ADS achieves linear speedups.

1.10 MODUS OPERANDI

Four main elements achieve ADS [17]:

- A scheduler that generates parallel schedules.
- Server software implementing a distributed platform where actors execute in a distributed manner.
- A message passing mechanism allowing the exchange of data over the network among actors in a decentralized manner.
- A centralized dispatching and synchronization mechanism that issues commands in parallel to exploit parallelism.

We have implemented ADS on top of the existing SDF MoC in Ptolemy, what we call Distributed SDF. To perform distributed simulations of existing SDF models it is sufficient to replace the SDF director by our Distributed-SDF director implementation without further modifications. A director is the component that implements the operational semantics to execute a model. We have added options (in the form of parameters in the configuration dialog) to generate sequential or parallel schedules and to execute the simulation in a local or distributed manner. We implement pipelining techniques as a major contribution to allow sequential topologies to benefit from distributed simulation.

There exist two main approaches for distributed simulation: SRIP (Single Replication in Parallel) and MRIP (Multiple Replication in Parallel). The former uses the fact that the process of solving a problem usually can be divided into smaller tasks, which may be carried out simultaneously with some coordination. In this case the tasks run on different processors. The latter consists in launching multiple runs of independent sequential simulations in parallel to different processors. Our implementation follows a SRIP approach taking advantage of the task partition, encapsulation and reusability already provided by the actors in Ptolemy models.

The implementation has been included as a new feature of the latest public release of Ptolemy II [22].

1.11 ASSESSMENT METHODOLOGY

The assessment of the ADS implementation requires cautious comparison with the original implementation. Due to the complexity and vast variety of models that can be implemented under the SDF domain we have experimented with several to assess performance. Models representing abstract version of real applications like a MPEG video encoder have been used. Actors that model the amount of time required to perform a task are used for accurate measurement of performance. A major factor providing gain from parallel execution is the *topology* of the model. Graphs, however, come in many shapes and we cannot consider all of them, because the number of experiments then grows exponentially in the number of actors. Therefore we have chosen two extreme cases and an intermediate one in terms of inherent parallelism. The two other factors we want to investigate are less complex: The *number of blocks* (actors), and the *number of iterations* of the model. Furthermore, we investigate the effects of granularity of actors on the results.

ADS yields smaller simulation times for any model where the cost of the most expensive component is greater than the cost of communication. Pipelining techniques allow any topology (especially purely sequential) to benefit from distributed simulation. Memory limitations are also overcome by distributing the memory consumption over several machines.

1.12 RELATED WORK

Related work includes papers describing distributed actor languages systems such as [18] and [19]; but they do not consider automatic distribution of actors. Some other efforts are being made towards distributing actor-oriented design models within Ptolemy; a proposal for a distributed approach is described in [20]. Contrary to our approach, it basically replicates the model in all the distributed processing elements. The overall goal of the Kepler project [21] is to produce an open-source system that allows scientists to design scientific workflows and execute those efficiently using emerging Grid-based approaches to distributed computation. It builds on top of Ptolemy and therefore is based on the same principles. Both approaches are in an initial stage and under development, so a closer comparison is not feasible at the moment. Our work is operational and has been included in the current Ptolemy II release [22]. Further related work concerning scheduling is included in Section 4.8.

1.13 THESIS OUTLINE

The remaining chapters of the thesis are organized as follows:

Chapter 2: outlines, describes and motivates Ptolemy II as our choice for the framework. It constitutes a summary of the documentation of the tool itself, its web page and various papers. The implementation of the actor formalism is also described as well as the architectural features of the tool relevant for this work. Its purpose is to

give the reader not familiar with the tool information we consider relevant to better understand this thesis. A reader familiar with the tool might want to skip to the next chapter.

Chapter 3: outlines the different computational models that are a stable part of Ptolemy II. Different relevant information is gathered and summarized in order to give an overview with a special emphasis on the different scheduling and dispatching mechanisms. The purpose is to motivate the choice of computational model and to discuss possibilities for distribution.

Chapter 4: surveys parallel scheduling and previous scheduling work (with emphasis on parallel scheduling) within the Ptolemy project. It describes the existing SDF scheduling paradigm and the approach taken for the implementation of ADS in the Distributed-SDF domain. This chapter provides details necessary for the following chapters.

Chapter 5: describes the different elements that achieve ADS and their implementation in Ptolemy II. Lower lever details of how ADS materializes and integrates with Ptolemy are described. This implementation has been included as a new feature in the official release of Ptolemy II [22].

Chapter 6: investigates the performance of the ADS implementation in Ptolemy II. Different experiments reveal the effect of relevant parameters. Results are analyzed and discussed in this chapter. Enabling pipelining techniques in ADS makes results for common applications insensitive to the topology of the models.

Chapter 7: draws conclusions and discusses future research topics.

1.14 PREVIOUSLY PUBLISHED RESULTS

The following articles have been published during the Ph. D. study:

- D. Lázaro Cuadrado, A. P. Ravn, P. Koch, "Automated Distributed Simulation in Ptolemy II", Parallel and Distributed Computing and Networks (PDCN 2007) proceedings, Acta Press.
- D. Lázaro Cuadrado, P. Koch, A. P. Ravn, "Using AsmL as an executable specification Language: Synchronous Dataflow Schedulers in Ptolemy II", *Workshop on Abstract State Machines ASM 2004*, May, 2004, Short Presentations Volume, Pages 19-28
- D. Lázaro Cuadrado, P. Koch and A. P. Ravn, "AsmL Specification of a Ptolemy II Scheduler ", *Lecture Notes in Computer Science, ASM 2003 Proceedings. Advances in theory and practice*, March, 2003, Vol. 2589

Conference presentations:

- D. Lázaro Cuadrado, A. P. Ravn, P. Koch, "The Distributed-SDF Domain", *The Sixth Biennial Ptolemy Miniconference*, May, 2005.
- D. Lázaro Cuadrado, P. Koch, A. P. Ravn, "Efficient and Orderly Co-simulation of Heterogeneous Computational Models", *EECS Department, University of California, Berkeley. The Fifth Biennial Ptolemy Miniconference*, May, 2003.

The implementation result of this work is publicly available as a new feature of the latest public release of Ptolemy II [22]. Since it is open source, it constitutes a major mean of dissemination of the ideas hereby presented, as well as documentation of this work.

The following articles and reports were published or filed on embedded systems topics. Though they are not directly related with this work, they gave a closer insight to the embedded systems world and great inspiration; therefore they are mentioned.

- D. Lázaro Cuadrado, O. Olsen, P. Koch, A. Frederiksen, O. Wolf R. D. Christiansen, "A Platform-Based Comparison Between A Digital Signal Processor And A General-Purpose Processor From An Embedded Systems Perspective", *NORSIG 2002 CD-ROM Proceedings,October, 2002*
- D. Lázaro Cuadrado, "3GDSP Final Report", *Aalborg University, Technical Report*, 2003, ISSN: 0908-1224 R03-1009
- D. Lázaro Cuadrado, A.Frederiksen, M. Genutis, "3GDSP Platforms Analysis", *Aalborg University, Technical Report*, 2003, ISSN: 0908-1224 R03-1001
- D. Lázaro Cuadrado, A. Frederiksen, "CVSD Case Study", *Aalborg University, Technical Report*, 2003, ISSN: 0908-1224 R03-1006
- D. Lázaro Cuadrado, A. Frederiksen, "Dhrystone Case Study", *Aalborg University, Technical Report*, 2003, ISSN: 0908-1224 R03-1004
- D. Lázaro Cuadrado, A. Frederiksen, "Matrix Case Study", *Aalborg University, Technical Report*, 2003, ISSN: 0908-1224 R03-1003
- D. Lázaro Cuadrado, J. P. Holmegaard, "Viterbi Case Study", *Aalborg University, Technical Report*, 2003, ISSN: 0908-1224 R03-1002

CHAPTER 2

PTOLEMY

Everything that is hard to attain is easily assailed by the generality of men.
Ptolemy (Tetrabiblos)

This chapter outlines, describes and motivates Ptolemy II as our choice for the framework. It constitutes a summary of the documentation of the tool itself, web page and various papers. The implementation of the actor formalism is also described, as well as the architectural features of the tool relevant for this work. Its purpose is to give the reader not familiar with the tool pertinent information to better understand this thesis. A reader familiar with the tool might want to skip to the next chapter.

2.1 PTOLEMY PROJECT

The Ptolemy project studies heterogeneous modeling, simulation and design of concurrent, real-time, embedded systems [23]. It has been carried out in the University of California at Berkeley for more than a decade (it started in 1989 [24]). The project borrows its name from Claudius Ptolemaeus, the second century mathematician, astronomer and geographer.

2.2 PTOLEMY OBJECTIVES

The Ptolemy project focuses on embedded systems, particularly those mixing interdisciplinary technologies, for example analog and digital electronics, hardware, software and mechanical devices. The aim is to model and simulate the embedded software together with the physical substrate on which it runs and the devices through which it interacts with the environment. Typically such environment cannot wait as opposed to interactive systems which are able to synchronize with it. These are called reactive systems [25].

The focus is on complex mixtures of diverse functionalities such as signal processing, feedback control, sequential decision making and user interfaces. The work is carried under the premise that no single general-purpose MoC is likely to emerge in the near future that will deliver what designers need. Moreover, a general

purpose framework would fail in accurately modeling the widely different functionalities and their interaction. Accuracy comes with imposing constraints resulting in a more specialized MoC unlikely to embrace any complex system. Instead, the Ptolemy project takes an approach that mixes heterogeneous MoCs, while preserving their distinct identity. It is realized by including formal MoCs in a software laboratory for experimenting with heterogeneous modeling [26]. This allows the designer to choose the MoC (or the subset of them) that best suits the target application, avoiding over- or under- specified models. Furthermore, it acknowledges the strengths and weaknesses of each MoC. The choices of MoCs strongly affect the quality of a system design. Yet, this approach poses new challenges as it has to define the semantics of the interaction of components governed by different MoCs. This is a non trivial problem area addressed by the project. Moreover, the mixtures of MoCs, convey to the designers the responsibility to cope with heterogeneity, which is alleviated by providing sophisticated, visual user interfaces, see Section 2.8.

The project proposes actor-oriented design consisting in the assembly of concurrent components in the form of actors. Actors provide well defined interfaces and thereby strong encapsulation. A disadvantage of component models is the overhead of run-time component interfaces. Ptolemy implements a framework that defines what it means to be a component, in particular an actor. What the framework knows about the components and the components know about each other is made explicit by defining interfaces. It provides a selection of protocols for the components to interact materialized in implementations of MoC called domains. Moreover it provides a lexicon for the components interaction as a type system. Furthermore, Ptolemy represents time, an abstraction that belongs to the physical world. This is materializes in different ways in the different computational models. It can be *real time* that advances uniformly, either discrete or continuously, or time can be a partial order, sorted by constraints imposed by causality among events.

2.3 PTOLEMY APPROACH & ADVANCES

The current technologies used in tackling the objectives are listed on the project's webpage as follows:

- Use of programming language concepts such as semantics, type theories, reflection, and concurrency theories in system-level design of electronic systems.
- Focus on domain-specific modeling and design problems so the designer can focus on the problem, not the tools.
- Emphasis on *understanding* of systems, which is promoted by visual representations, executable models, and verification.
- Use of Java, design patterns, UML, and a modern software engineering practice adapted to the realities of academic research.

Further current, experimental and future capabilities can be found under sections 1.5.6-8 in [26].

2.4 CURRENT HOT RESEARCH AREAS

Hot research topics that are currently being addressed in the project include:

- **Precision-timed (PRET) machines**: This effort reintroduces timing into the core abstractions of computing, beginning with instruction set architectures, using configurable hardware as an experimental platform.
- **Real-time software**: Models of computation with time and concurrency, metaprogramming techniques, code generation and optimization, domain-specific languages, schedulability analysis, programming of sensor networks.
- **Distributed computing**: Models of computation based on distributed discrete events, backtracking techniques, lifecycle management, unreliable networks, modeling of sensor networks.
- **Understandable concurrency**: This effort focuses on models of concurrency in software that are more understandable and analyzable than the prevailing abstractions based on threads.
- **Systems of systems**: This effort focuses on modeling and design of large scale systems, those that include networking, database, grid computing, and information subsystems. For example the Kepler project which targets scientific workflows.
- **Abstract semantics**: Domain polymorphism, behavioral type systems, meta-modeling of semantics, comparative models of computation.
- **Hybrid systems**: Blended continuous and discrete dynamics, models of time, operational semantics, and language design.

2.5 ACTORS AND COMPONENTS

Ptolemy II focuses on component-based design and this is carried out by the use of actors. We include the definition for clarity:

Szyperski's definition of software component [27]:

> 'A software component is unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.'

An actor is often described in terms of its structure and operation. It was originally introduced by Hewitt in 1973 [28] and further developed by Agha [29]. An actor is a message processing entity that receives incoming messages. An actor's reaction to a

message is determined by the actor's behavior when the message is processed (send messages to others, create new actors, determine its own subsequent behavior, etc).

From these definitions, and actor is a component. Throughout the following we will make use of both and use component for the more general picture and actor for the more specific concept in Ptolemy.

2.6 ACTOR-ORIENTED DESIGN

Actor-oriented design [30] is a system-level design methodology that relies on the actor model. The actor model is a model of concurrent computation that can be used as a framework for modeling, understanding, and reasoning about, a wide range of concurrent systems. It was first introduced by Hewitt in 1973.

The actor model treats actors as the universal primitives of concurrent digital computation, adopting the philosophy that everything is an Actor. This is similar to the philosophy used by some object-oriented programming languages where everything is an object. The difference is that what flows through an object is sequential control whereas data flows through actors. Simula inspired both object-oriented programming languages and the actor model by supporting co-routines but not true concurrency. The actor model has been used both as a framework within which to develop a theoretical understanding of concurrency, and as the theoretical basis for several practical implementations of concurrent systems. Unlike in Linda-like systems, the exchange of data is conveyed among actors without an intermediate shared memory space.

An Actor is a computational entity with a behavior such that in response to each message received, it can concurrently:

- make local decisions
- send a finite number of messages to (other) actors
- create a finite number of new actors
- designate the behavior to be used for the next message received

There is no presumption on the execution order for the actions above which could in fact be performed in parallel. Thus arises two types of concurrency, inter- and intra-actor. We take advantage of the inter-actor concurrency for distribution. The intra-actor concurrency is preserved intact, this way avoiding violating encapsulation, since its exploitation would require knowledge of the current actor implementation.

Communications with other actors can occur both synchronously or asynchronously depending on the MoC that governs the interaction. In asynchronous communication, the sending actor does not wait until the message has been received before proceeding with its computation. Both have to be ready in the synchronous what is called *rendezvous*. Messages are sent to specific actors, identified by an address (sometimes referred to as the actor's mailing address). As a result, an actor can

only communicate with actors for which it has an address, which it might obtain in the following ways:

- The address is in the a received message
- The address is one that the actor already had, it was already an acquaintance
- The address is for an actor created by this actor

In Ptolemy, the actor addresses are specified by a model specific topology. The actor model is characterized by inherent concurrency of computation within and among actors, dynamic creation of actors and acquaintance relationships, inclusion of actor addresses in messages, and interaction only through direct asynchronous message passing with no restriction on message arrival order. However, the SDF implementation preserves the order for a sequence of messages between two actors.

In actor-oriented design, actors are the primary units of functionality. They have a well defined interface, which abstracts internal state and restricts interaction with its environment. Actors are composed to form models and the semantics of the composition are given by a MoC. Connections between the actor ports represent communication channels that pass data tokens from one port to another. A central concept in actor-oriented design is that internal behavior and state of an actor are hidden behind the actor interface and not visible externally. This property of strong encapsulation separates the behavior of a component from the interaction of that component with other components. System architects can design at a high level of abstraction and consider the behavioral properties of different models of computation independently from the behavioral properties of components. By emphasizing strong encapsulation, actor-oriented design addresses the separation of concerns between component behavior and component interaction.

The actor-oriented models implemented in Ptolemy are different from the actor model description in that they generally have a static structure given by a fixed topology while still allowing dynamic instantiation. Furthermore, Ptolemy introduces typed data tokens that encapsulate a value of a specific type and are used to convey data among the actors. This distinction allows the optimization of execution performance with respect to the static structure of a model without considering dynamic allocation of space for tokens [31].

One of the advantages of actors is reusability, which enhanced in Ptolemy since actors in Ptolemy can adapt to the data (polymorphism).

The actor abstraction exposes the available functionality to the outside world in a standardized way common to all actors. This is materialized by all actors and directors implementing the `Executable` interface. This interface defines *action* methods that can be used by the directors to orchestrate execution:

- `preinitialize()`: is invoked only once before any other use. Prior to any static analysis performed on the model (scheduling, type inference, deadlock checking).
- `initialize()`: is invoked once after type resolution. It initializes the actor or director to its initial state.

The following can be invoked several times:

- `prefire()`: should be invoked before an invocation of `fire()` to check whether it is ready, for example to check availability of data tokens.
- `fire()`: may be invoked several times between invocations of `prefire()` and `postfire()`. This method defines the computation performed by the actor; reading of input data and producing output data.
- `postfire()`: should be invoked once after the last invocation of `fire()`. Here the actor may update its local state.

One invocation of `prefire()`, followed by any number of `fire()` and a `postfire()` defines an iteration. The `iterate(int count)` action method is provided for easy use, where count corresponds to the number of times `fire()` has to be fired in the iteration.

- `wrapup()`: should be invoked once per actor to finalize execution of a model. It is responsible for cleaning up after execution has completed.

One invocation of `preinitialize()`, followed by `initialize()` plus a number of iterations and an invocation of `wrapup()` define an execution.

Different types of actors are documented in Chapter 4 in [26].

Because the internal state of each actor is hidden from other actors in a model, there are few intrinsic constraints on the firings of different actors in a model. The lack of execution constraints implies that actors in a model are fundamentally concurrent. In order to obtain efficient execution in single processor environments, additional execution constraints are often added to a model to enforce sequential execution of actor firings. In the absence of additional constraints, the firings of two distinct actors are allowed to occur completely independently. This implicit concurrency is exploited in this work.

2.7 MESSAGE PASSING

Message passing is a form of communication used in concurrent programming and in particular for the actor model. Communication is exchange of messages. These messages convey some form of information. Different types of messages include function invocation, signals, and data packets. This concept is also used for example in microkernel OS systems, and to implement distributed objects, remote method invocation (RMI) systems like RPC, CORBA, Java RMI, DCOM, SOAP. The term is

also used in High Performance Computing (HPC), in the form of Message Passing Interface (MPI). Message passing systems hide underlying state changes that may be used in the implementation of sending messages thus encapsulating the network carrying the messages.

The exchange of messages may be carried out asynchronously where the sender does not have to wait for the receiver to continue with its computation. Message exchange can also be arranged using rendezvous, where both sender and receiver synchronize to communicate.

In the Ptolemy implementation, both synchronous and asynchronous message passing mechanisms are used in the different MoC. Often when using asynchronous message passing, unbounded buffers are used. Concurrency is easier to reason about when using synchronized message passing exchange or asynchronized with unbounded buffers.

Messages are encapsulated in tokens and exchanged via ports. Input ports contain `Receivers` capable of receiving messages from distinct channels.

The semantics of the composition, including the communication style is determined by a model of computation. When necessary, the model of computation will be shown explicitly as an independent director object in the model. Models often export an external actor interface, enabling them to be further composed with other models. Data transport details can be found under Chapter 2 in [40].

2.8 GRAPHICAL SYNTAXES

Among the different ways to construct models using the tools provided by the Ptolemy project, there is special focus on visual rendering of systems. Diagrams allow designers to deal with graphical or iconic elements that can be manipulated spatially in an interactive way rather than text. They provide the ability to make explicit the concurrent activities and distributed structure of the target systems. Ptolemy proposes an abstract syntax of clustered graphs combined with textual annotations for precision.

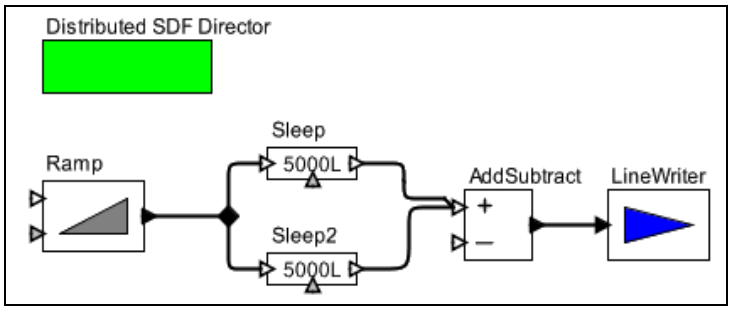


FIGURE 2: VISUAL SYNTAX IN PTOLEMY II

Figure 2 shows the visual syntax of an example model and Figure 3 shows the XML representation of the same model. The visual rendering provides better readability over the XML therefore conveying information about the model faster.

```
<?xml version="1.0" standalone="no"?>
```

```
<!DOCTYPE entity PUBLIC "-//UC Berkeley//DTD MoML 1//EN"
    "http://ptolemy.eecs.berkeley.edu/xml/dtd/MoML_1.dtd">
<entity name="m2t" class="ptolemy.actor.TypedCompositeActor">
    <property name="_createdBy" class="ptolemy.kernel.attributes.VersionAttribute" value="5.1-alpha">    </property>
    <property name="_windowProperties" class="ptolemy.actor.gui.WindowPropertiesAttribute" value="{bounds={-745, 280, 734, 536},
maximized=false}">    </property>
    <property name="_vergilSize" class="ptolemy.actor.gui.SizeAttribute" value="[519, 418]">    </property>
    <property name="_vergilZoomFactor" class="ptolemy.data.expr.Parameter" value="1.0">    </property>
    <property name="_vergilCenter" class="ptolemy.data.expr.Parameter" value="{236.2474747474748, 163.155303030303}">    </property>
    <property name="Distributed SDF Director" class="ptolemy.distributed.domains.sdf.kernel.DistributedSDFDirector">
        <property name="iterations" class="ptolemy.data.expr.Parameter" value="20">        </property>
        <property name="parallelSchedule" class="ptolemy.data.expr.Parameter" value="true">        </property>
        <property name="pipelining" class="ptolemy.data.expr.Parameter" value="false">        </property>
        <property name="parallelExecution" class="ptolemy.data.expr.Parameter" value="true">        </property>
        <property name="_location" class="ptolemy.kernel.util.Location" value="{80.0, 10.0}">        </property>
    </property>
    <entity name="Ramp" class="ptolemy.actor.lib.Ramp">
        <doc>Create a sequence of tokens with increasing value</doc>
        <property name="_location" class="ptolemy.kernel.util.Location" value="[40.0, 95.0]">        </property>
    </entity>
    <entity name="LineWriter" class="ptolemy.distributed.actor.lib.DistributedLineWriter">
        <doc>Outputs a line with the incoming token in the standard output.</doc>
        <property name="_location" class="ptolemy.kernel.util.Location" value="[375.0, 100.0]">        </property>
    </entity>
    <entity name="Sleep" class="ptolemy.actor.lib.Sleep">
        <property name="sleepTime" class="ptolemy.actor.parameters.PortParameter" value="5000L">        </property>
        <property name="_icon" class="ptolemy.vergil.icon.BoxedValueIcon">
            <property name="attributeName" class="ptolemy.kernel.util.StringAttribute" value="sleepTime">            </property>
            <property name="displayWidth" class="ptolemy.data.expr.Parameter" value="40">            </property>
        </property>
        <property name="_location" class="ptolemy.kernel.util.Location" value="[170.0, 65.0]">        </property>
    </entity>
    <entity name="AddSubtract" class="ptolemy.actor.lib.AddSubtract">
        <property name="_location" class="ptolemy.kernel.util.Location" value="[280.0, 100.0]">        </property>
    </entity>
    <entity name="Sleep2" class="ptolemy.actor.lib.Sleep">
        <property name="sleepTime" class="ptolemy.actor.parameters.PortParameter" value="5000L">        </property>
        <property name="_icon" class="ptolemy.vergil.icon.BoxedValueIcon">
            <property name="attributeName" class="ptolemy.kernel.util.StringAttribute" value="sleepTime">            </property>
            <property name="displayWidth" class="ptolemy.data.expr.Parameter" value="40">            </property>
        </property>
        <property name="_location" class="ptolemy.kernel.util.Location" value="[170.0, 125.0]">        </property>
    </entity>
    <relation name="relation4" class="ptolemy.actor.TypedIORelation">    </relation>
    <relation name="relation" class="ptolemy.actor.TypedIORelation">
        <property name="width" class="ptolemy.data.expr.Parameter" value="1">        </property>
        <vertex name="vertex1" value="[120.0, 95.0]">        </vertex>
    </relation>
    <relation name="relation2" class="ptolemy.actor.TypedIORelation">
        <property name="width" class="ptolemy.data.expr.Parameter" value="1">        </property>
    </relation>
    <relation name="relation3" class="ptolemy.actor.TypedIORelation">
        <property name="width" class="ptolemy.data.expr.Parameter" value="1">        </property>
    </relation>
    <link port="Ramp.output" relation="relation"/>
    <link port="LineWriter.input" relation="relation4"/>
    <link port="Sleep.input" relation="relation"/>
    <link port="Sleep.output" relation="relation2"/>
    <link port="AddSubtract.plus" relation="relation2"/>
    <link port="AddSubtract.plus" relation="relation3"/>
    <link port="AddSubtract.output" relation="relation4"/>
    <link port="Sleep2.input" relation="relation"/>
    <link port="Sleep2.output" relation="relation3"/>
</entity>
```

FIGURE 3: MOML VERSION OF THE MODEL SHOWN IN FIGURE 2

2.9 TOOLS

Several tools have been released throughout the years of existence of the Ptolemy project. We introduce them below and highlight their main features. They constitute the software infrastructure that has implemented the research advances creating a laboratory for the experimental side of the Ptolemy project.

The Ptolemy environment has been used for a broad range of applications including signal processing, telecommunications, parallel processing, wireless communications, network design, investment management, modeling of optical communication systems, real-time systems, and hardware/software co-design among others [24][32][35][36][37][38]. Ptolemy software has also been used as a laboratory for signal processing and communications courses. Currently Ptolemy software has

hundreds of active users at various sites worldwide in industry, academia, and government.

The first software developed was called Gabriel, later mainly two tools have been developed. The first one is called Ptolemy Classic and its descendant is Ptolemy II.

2.9.1 GABRIEL

Already in the late 1980s Gabriel [32] provided a graphical interface to describe algorithms with dataflow diagrams. It is an evolution of a simulation tool called Blosim [33]. It aimed at the development of assembly and microcode for programmable DSPs and was written in Lisp. The function blocks are named stars, already encouraging encapsulation and code reusability. Hierarchical design is also possible by means of galaxies, clusters of stars. The SDF diagrams matured under the construction of Gabriel. Parallel scheduling techniques were implemented to improve the run time for the stars. Efficient code generation was provided for selected processors.

2.9.2 PTOLEMY CLASSIC

Ptolemy Classic was the first heterogeneous simulation and design environment supporting multiple models of computation. Development started in 1990. It has a graphical user interface for constructing models visually as block diagrams [23][24]. The Ptolemy project continues to use Ptolemy Classic occasionally as a laboratory for experimenting with models of computation, but most of the work has shifted to Ptolemy II. Ptolemy Classic is a platform for simulation and rapid prototyping based on Gabriel. The latest stable release is 0.7.1, from 1998, moreover, there exist a newer release available for testing 0.7.2 for experienced Ptolemy developers. Ptolemy Classic is flexible and extensible, and allows mixed hardware and software designs and prototyping by mixing MoC. Enabling heterogeneous simulations is the main breakthrough. The supported computational models are SDF, DDF, DE, PN, FSM and SR [34]. The programming language was changed from Lisp to C++ for computational efficiency, integration of DSP algorithms and other code written in C and wider availability of compilers and tools. Matlab and VHDL code is also permitted for implementation. The system can generate C code and assembly for at least two programmable DSPs (TI TMS320C50 and Motorola DSP56000) and VHDL for certain dataflow descriptions of systems. Commercial EDA tools like Agilent ADS and Cadence's SPW included parts or were inspired by methods developed in Ptolemy Classic.

The different components in Ptolemy Classic encourage users to build reusable software components:

- *Block*: Is a module of code, which is invoked at run time. There are several types of blocks:

  - *Star*: Is an atomic block. Typically performs some unit of computation during de simulation.
  - *Galaxy*: is a block that internally contains a collection of interconnected Blocks. It allows hierarchical descriptions of blocks and to partition designs into re-usable units that can be used as components in other designs.
  - *Universe*: Outermost block, which contains the entire simulation. Internally it consists of a galaxy and an associated scheduler.
  - *WormHole*: is a star that internally contains a universe.
- *Scheduler*: Determines a block's execution order (either a compile time, runtime or both).
- *PortHole*: A block's interface to communicate with other blocks.
- *Particle*: A message sent between blocks.
- *Domain*: The combination of a scheduler and a set of blocks that obey a common computational model.

2.9.3 PTOLEMY II

Ptolemy II is the current software incarnation of the Ptolemy Project, actively under development. It is based on and an evolution of its predecessor Ptolemy Classic. Ptolemy II is a graphical design tool for hierarchical and component based system models that provides a platform for simulation and rapid prototyping.

A model in Ptolemy consists of a set of actors, a topology that interconnects them and a MoC defined by a director. The topology is represented by hierarchical clustered graphs. (This does not apply to all the computational models (for example FSM) but to the majority).

The director dictates how and when the actors execute and interact with each other, defining the behaviour of the model, the semantics. This set of rules constitutes the MoC under which the model runs and it is called domain in Ptolemy II. Vergil [26] is a front-end graphical user interface that allows users to create block diagrams that visually represent Ptolemy models. Users can drag and drop components, connect components, edit component parameters, and execute models. The visual syntax of Vergil is very appealing because model creation feels more like drawing than programming. Ptolemy II supports the construction and interoperability of executable models that are built under different MoCs, therefore it can be regarded as a laboratory for exploring MoCs. Model execution allows observing system design behavior at runtime. Ptolemy II follows the actor oriented design approach thus stressing concurrency. The basic unit of computation is the actor. Actors provide modularity since they constitute individual entities offering a set of features. Actors interact with each other forming a model of a system. Such interactions take place by exchanging

data tokens through ports. Well-defined interfaces dictate how the actor communicates with the rest of the model, abstracting away the internal workings. These interfaces consist of ports which represent points of communication and parameters that allow adjusting execution. Actors may consume tokens from input ports, produce tokens on output ports during execution. Functionalities may include creating visual output on the screen, accessing files or even producing audio output among others. Actor execution is also called firing. Many actors have customizable parameters, making them quickly configurable and promoting reusability. Component-based design in Ptolemy II involves disciplined interactions between components governed by a model of computation.

A very powerful abstraction provided by Ptolemy II is that of hierarchy, a *composite actor* is an actor that can contain other actors working together to deliver some coherent functionality or even containing a complete model within it. To provide a layer of abstraction, composite actors also have ports and parameters that can be connected from the inside to any actor within. Actors at the same level can be connected. To them, the composite is seen as a black box, and its behaviour can be analyzed without understanding the contained details. This makes composite actors domain polymorphic allowing them to execute under different top-level domains.

A composite actor can also contain a Director defining a different Moc. This allows different parts of a model to operate under different MoCs thus providing heterogeneity. Composite actors constitute an improvement from the preceding Ptolemy Classic's wormhole mechanism, permitting an easier mechanism to combine dataflow and non-dataflow MoC. In the absence of a Director, the parent model's Director governs execution. Composite actors can be arbitrarily nested thus yielding hierarchy and heterogeneity.

Some actors in Ptolemy II are *domain specifi*c, meaning that they have been designed to function in one domain; therefore certain assumptions are made related to the domain. These actors will not function correctly in domains that do not satisfy the assumptions. The *domain polymorphic* actors are those which will work in as many domains as possible thus promoting reusability.

Depending on the type of tokens actors recognize, they can be classified as *data-type specifi*c or *polymorphic*. Data specific actors only operate on certain types of tokens whereas *data polymorphi*c can operate on any of a number of types of tokens. The behaviour of the latter may be different depending on the type of input data received. For example, the `AddSubstract` actor adds numbers and concatenates strings. Ptolemy II uses a new algorithm to determine the types of ports in a system. The algorithm uses a partial order of types and monotonic type refinement [39].

Ptolemy II provides coordination among actors without knowledge of the functionalities they implement, achieving high-level concurrency models that can be used by non experts.

Ptolemy II is written in Java which makes it platform-independent, threaded, network aware, allowing execution of code from remote sources securely. There is one installation that runs entirely on applets. Java's automatic garbage collection spares programmers from having to perform manual memory management.

The latest release is 6.0.2 from February 2007, though a development version that is constantly evolving is accessible via CVS.

2.9.3.1 Organization of Ptolemy II

Ptolemy II provides a unified infrastructure in a collection of Java packages which can be used independently providing extra modularity. The software architecture was designed following object modeling and using design patterns. Abstract syntax is defined in the software kernel, consisting of classes that build up a hierarchical graph. Abstract semantics are implemented in the different packages that implement the different computational models. The key packages are introduced in the following:

- *Kernel*: implements a data structure that supports a generic form of clustered hierarchical graphs. In this context, graphs are sets of entities and relations providing a general *abstract syntax*. Entities have ports that interconnected through relations in a multiway manner. Hierarchy is achieved by nesting encapsulated graphs inside composite entities.
- *Actor:* Provides support for executable actors and the infrastructure for concurrency. It extends entities in the kernel so that entities have functionality and can communicate via the relations. Infrastructure in the form of templates is provided for message passing. Actors can communicate with another if it can obtain a reference to a receiver in an input port of the latter. This mechanism is provided by relations. Data transport does not involve relations allowing for example global references to receivers. A subpackage contains a library of polymorphic actors.
- *Domain packages*: extend the actor package by imposing MoC computation on the interaction between entities. These are further described in Chapter 3.
- *Data:* implements a type system providing data encapsulation, polymorphism, parameters and an expression language. Data encapsulation is implemented by a set of token classes with a standard interface. The tokens are immutable meaning that the value cannot be modified. The base token defines methods for primitive arithmetic operations such as `add()`. The values of parameters can be specified directly or by expressions that may refer to other parameters. The expression language allows scripting without compilation.
- *Graph*: provides graph-theoretic manipulations. Ptolemy graphs implement hierarchy and multiway connections. Support for directed and undirected graphs is provided. A subpackage provides infrastructure for schedules both sequential and parallel.

- *Math*: providing matrix and vector math and signal processing functions.
- *Plot*: provides infrastructure for visual display of data.
- *Codegen*: Template based code generator similar to Ptolemy Classic.
- *Copernicus*: Code generator that opposed to codegen reuses the existing Java code in the actors to produce executable prototypes that are smaller and faster.
- *Matlab:* uses the Java Native Interface to invoke the Matlab engine, evaluate Matlab expressions and return the results to Ptolemy II.

2.9.3.2 Documentation of Ptolemy II

For further and more in depth description of the Ptolemy II tool, three volumes are provided under the common title "Heterogeneous Concurrent Modeling and Design in Java":

- *Volume 1: Introduction to Ptolemy II* [26]: It can be seen as a user manual describing the use of the software. Ptolemy is introduced and includes some tutorias describing how to build models and applications
- *Volume 2: Ptolemy II Software Architecture* [40]: Its target audience is developers. Describes the software architecture and the packages implementing different features.
- *Volume 3: Ptolemy II Domains* [41]: Describes the implementation of a MoC in each domain.

CHAPTER 3

MODELS OF COMPUTATION IN PTOLEMY II

Real-time software has done amazingly well using the wrong foundational abstractions. Just think what we could do with the right ones!
Edward A. Lee

Different domains that compound a stable part of Ptolemy II are described in this chapter. They are implementations of models of computation useful for embedded systems. Each of them deals with concurrency and time in a different way. Support for modeling time and concurrency is also detailed. Diverse relevant information is gathered and summarized in order to give an overview with a special emphasis on the different scheduling and dispatching/coordination mechanisms. The purpose is to motivate the choice of computational model and to discuss possibilities for distribution.

A model of computation governs component interactions, hence providing the semantics that determine the behavior of a model. Behavior is defined by the model's state changes and interaction among components. This is mainly achieved by two elements: an *execution mechanism* that sorts the execution of components and a *communication mechanism* that defines how component interaction occurs. If there is a notion of *time*, the MoC must define how it progresses. Models of computation that handle concurrency and time are best suited for embedded systems. Ptolemy II domains implement MoCs in a way that allows the separation of the (nearly) common abstract syntax of models, from their distinct semantics. Actors do not need to be aware of the communication mechanism in their implementation. They call methods of ports to communicate, but the domain specifies the behavior, enabling and facilitating the development of polymorphic actors.

Models of computation may exist at all levels of abstraction in a system. The MoCs implemented in Ptolemy II provide a high level of abstraction for design. They belong to the specification level, providing a platform for design from above and the the same time hiding the underlying implementation details below.

3.1 EXECUTION MECHANISMS

Two different execution approaches for models are implemented in Ptolemy II, *schedule* and *process* based [42]:

3.1.1 SCHEDULE BASED

Schedule-based execution presumes that each component's computation is finite. In this approach, a schedule is created that specifies an ordering of execution of actors in the model. Schedules in Ptolemy II are sequential, but parallel schedules are also supported. Parallel schedules expose the inherent parallelism of models. Domains like CT, DE or SDF are schedule based.

The package `ptolemy.actor.sched` provides generic infrastructure for schedule based domains. A more detailed description of this package and scheduling techniques is given in Chapter 4.

3.1.2 PROCESS BASED

The process-based approach does not assume that an actor's execution is finite. For the sake of fairness the process-based approach embeds each actor in a thread of control. Thus, each actor acts as a separate *process* running concurrently. Actors become autonomous in their execution, which is not controlled by the director. Domains like CSP, DDE, and PN are process based.

The package `ptolemy.actor.process` provides generic infrastructure for process based domains. The infrastructure that allows processes to call their actor's own action methods is provided by the `ProcessThread` class. This class is associated with `ProcessDirector` and `Actor`. An actor can be passed as an argument in the constructor and when started, it calls the action methods of the actor. To call termination, an actor returns false in its `postfire()` method, this causes the actor to never fire again and the process terminates. Directors need to keep track of the state of processes, whether they are active or blocked, in order to react to deadlock. This is provided in the `ProcessDirector` class which defines standard empty methods to detect and handle deadlock. These should be overridden by deriving classes for specific behaviors. `CompositeProcessDirector` adds support for hierarchical heterogeneity. The `ProcessReceiver` interface provides mechanisms to terminate simulation. In this context, simulations are normally ended by detecting a deadlock. Deadlocks occur when processes are waiting, normally for reading or writing.

The process based approach can be viewed as a high level abstraction for a concurrent design, which hides the implementation details. These abstractions are easier to use to design concurrent processes than threaded programs.

On the other hand, the process based approach usually results in less efficient execution than the schedule based due to the overhead introduced by coordinating threads dynamically.

Process based domains present good candidates for distribution by exposing concurrency of the models.

3.2 COMMUNICATION MECHANISMS

The following communication mechanisms appear in the different domains:

- *Asynchronous message passing*: When process A sends a message to process B, the message is stored in the process B's queue of messages. SDF, DE, DDE, PN are examples of domains that use this communication mechanism.
- *Synchronous message passing*: Here communicating processes have to synchronize. If one of the processes is not ready, it has to block until the other is ready. This is called a *rendezvous*. CSP implements this communication mechanism.

Receiver classes support these communication mechanisms.

3.3 MODELING OF TIME

One of the objectives of Ptolemy II is to represent time and simulate its progress. There are different notions of time that should not be confused:

- *Physical time*: time in the system being modeled.
- *Simulation time*: abstraction used in the simulation to model physical time.
- *Real time*: time during the execution of the simulation (also referred as wallclock time).

The class `ptolemy.actor.util.Time` implements the representation of simulation time providing support for quantization and comparison.

There are different ways to model time in a simulation. All of them are implemented in Ptolemy II, in one or several domains:

- *Time Slicing:* divides simulation time in time steps to track time progress. An important issue is determining the optimal value of the step. It is inefficient because in many of the time steps no system state changes occur, therefore wasting computation power. This approach is used in DT.
- *Discrete Event*: represents only the points in time where system state changes. Ptolemy uses this approach in most timed MoC, the most obvious is DE.
- *Continuous simulation*: Some systems experience changes in state continuously though time, good examples are signals in communication systems. Since digital computers cannot model continuous changes therefore

approximations are needed in this case. The CT MoC tackles this type of systems.

Ptolemy simulations can adjust the progression of time to real time (if they can catch up with it) to appear realistic, thus providing a real-time simulation. Despite the fact that most computational models implement a global notion of time that hinders its distribution, DDE implements one solution to this problem.

3.4 PTOLEMY DOMAINS

The various implementations of MoC in Ptolemy II are described here:

3.4.1 SYNCHRONOUS DATAFLOW (SDF)

The Synchronous Dataflow [43] domain supports the efficient execution of Dataflow graphs where the flow of control is predictable prior to runtime. This comes at the expense of imposing some restrictions on the graphs. Actors are required to declare the number of tokens they consume and produce, and this number remains constant throughout execution. Actors can fire when enough tokens are available at the input ports. Connections among the actors represent the flow of data and communication is performed via asynchronous message passing. In addition, feedback loops must be broken by delays.

SDF suits applications that handle regular computations that operate on streams. Digital signal processing applications are typically modeled with SDF [44]. Other tools implementing SDF are SPW and LabVIEW [45].

Some important properties arise in this MoC. Since rates and delays are known and fixed before runtime, it allows computing a static schedule [46]. The sizes of the queues that store incoming data at the ports are known and bounded. Iteration is defined as the minimal set of firings that return the queues to their original sizes. Deadlock is decidable. On the other hand, these constraints specialize SDF making it less suitable for some applications.

There are some variants of SDF in which the number of tokens produced and consumed need not to be constant [47].

- *Boolean dataflow (BDF)*: is a generalization that tries to compute a compile time annotated schedule. The annotations consist of boolean conditions on the firings. The boolean conditions are computed at runtime, thus resulting in less overhead than in a full dataflow language.
- *Dynamic dataflow (DDF)*: is a superset of SDF and BDF. In DDF an actor can change the production and consumption rates after each firing. The scheduler makes no attempt to construct a compile-time schedule; it uses only run-time analysis.

- *Heterochronous dataflow (HDF)*: Actors have a finite number of rate signatures specifying the number of tokens produced and consumed. They are allowed to change between iterations.
- *Cyclo-Static Data Flow (CSDF)*: Actors cycles through a finite list of rate signatures. Static scheduling is possible.

BDF and DDF are much more expressive than SDF, HDF or CSDF at the expense of loosing decidability. Experimental implementations of HDF and DDF are included in Ptolemy II. The implementation of the SDF domain generates purely sequential execution of models. No effort to exploit the inherent parallelism is made in the standard implementation.

3.4.2 DISCRETE EVENT (DE)

Discrete event simulation is one way of building up models to observe the dynamic time based behavior of a system. In discrete event simulation only the points in time at which the state of the system changes are represented. The system is modeled as a sequence of events. Each event occurs at an instant in time and marks a change of state in the system. Therefore simulation time is a totally ordered set of values each of them representing an instant in the system. This is contrary to continuous simulation where the system evolves as a continuous function. There are a large number of potential application areas; some examples include design of assembly lines, service industries and time-oriented models of systems such as queueing systems, communication networks and digital hardware. It specially suits describing concurrent digital hardware. It has been realized in a large number of simulation environments, simulation languages and hardware description languages including VHDL and Verilog [49].

In this domain, actors communicate by asynchronously sending *event*s, where an event is a data value (a token) and a *tag*. The tag is a tuple containing a natural number representing time (*time stam*p) and a number (*microstep)*. The time stamp is generated by the actor that produces the event using the time stamp of input events, and the latency of the actor. The microstep is used to index the sequence of execution for events with the same timestamp. For example, when an actor requests to fire again at the current model time a new event with the same timestamp is generated with an incremented microstep. If two events have the same tag they are called simultaneous events. The events are processed chronologically according to the tag by firing those actors whose available input events are the oldest (having the smallest timestamp of all pending events). In case of several events having the same timestamp, the order is dictated by the smaller microstep. In case of simultaneous events, *depth* allows to handle events in a deterministic way. Depth is a number that is obtained by topologically sorting all the ports and actors according to their data dependencies. The TimeDelay actor can be used to break loops to achieve a DAG for topological sort. If

timestamp, microstep and depth are the same, the events are called identical. Formal semantics for DE are given in [49].

There is a globally consistent notion of simulation time and microstep that is known simultaneously throughout the system and can only be modified by the director. The DE allows (attempting) synchronization to real time to make simulations more realistic. When a token is sent through an output port, it is packaged as an event and stored in a global event queue. This is called a *trigger* event with a port as a destination. If the destination of the even is an actor it is called a *pure* event. They do not contain data and they are used to fire the actor again at a given time, greater or equal to current time. Events are queued in the global event queue according to their timestamps, microsteps and depth of destination actors. The implementation uses a Calendar Queue [50], a fast priority queue, which provides constant complexity (O(1)) for the queue() and dequeue() operations for a large number of events. The global even queue is used as a scheduling mechanism by the director.

Due to global notion of time, a distributed version of DE poses a challenge that is addressed by DDE.

3.4.3 DISTRIBUTED DISCRETE EVENT (DDE)

DDE is used in modeling and designing systems that are naturally timed and distributed. Ptolemy implements a variant of DDE as in [51]. In order to overcome global time, the distributed discrete event (DDE) domain distributes the notion of time to each process. Each actor is controlled by a thread (process based), which maintains the local notion of time. Time progresses in a DDE model when the actors in the model execute and communicate. The DDE domain implements a conservative approach to preserve the processing of incoming messages in time stamp order (*local causality constraint*). Adherence to the causality constraint is sufficient to ensure that the distributed simulation will produce the same results as a sequential execution. Conservative approaches avoid violating the local causality constraint by waiting until its safe to continue as opposed to optimistic methods, which allow violations of the local causality constraint [52]. Tokens produced by a port must have an equal or greater time stamp that the preceding ones thus avoiding inconsistencies. Moreover, a static reliable network topology that delivers messages preserving the order is assumed to assure messages are received with ascending time stamp order. Whenever a queue empties, the process must block to prevent events with earlier timestamps that might have not arrived yet to be processed in the wrong order. This makes conservative approaches highly prone to deadlocks that are not real. This is resolved by sending extra tokens are sent through the ports for the only purpose of maintaining local times updated. The time stamp of such tokens is a lower bound for future incoming messages in the receiving actors. It is calculated as the current time of the process plus a look-ahead (for example the time the process takes to process an event). Moreover, actors notify terminating execution to the actors connected to its output

ports, including timestamps that are used to update their local times. Therefore local time information is equivalent to the maximum time stamps consumed by the actor. Similar to DE, prioritizing of ports takes place in case of simultaneous events. Tokens with the smallest time stamp for a given actor are consumed first. Feedback delays are used to break cycles in cyclic topologies.

In traditional DE systems, all the events are ordered in the global event queue whereas in DDE events are partially ordered, thus all the events associated with an actor are ordered. In DE total order of events forces sequentially execution whereas in the DDE approach, concurrency exposed by the topology can be exploited. This comes at the expense of extra complexities and traffic to maintain local times.

3.4.4 PROCESS NETWORKS (PN)

Kahn process networks [53] is a highly concurrent MoC where processes are connected by communication channels to form a network. There is no notion of time. Processes represent functions that map input streams into output streams. Streams of data are infinite; therefore communication takes place via (ideally) unbounded FIFO channels. Actors communicate via asynchronous message passing through blocking reads of FIFO queues. PN is a superset of SDF.

It can be applied to model distributed systems, hardware architectures and specially signal processing applications, therefore being well suited for embedded systems. Signal processing naturally matches the infinite streams of data which are the default communication medium in PN.

Variants of this model are used or have been used in commercial visual programming systems such as SPW form Cadence, COSSAP from Synopsys, DSP Station from Mentor Graphics, and Hypersignal from Hyperception (now integrated in LabVIEW). Also used in research software such as Khoros from the University of New Mexico among others [54].

Ptolemy II's implementation of PN is process based [55]; the director creates a thread for each actor in the model thus allowing all actors to execute concurrently. The topology represents the streams of data (encapsulated in tokens) exchanged between actors. An actor can read input tokens from input ports, and write tokens to output ports. Normally, when it reads from an input port, the read *blocks* until an input token is available. Actors cannot ask an input port whether there are available tokens to read, instead, they must simply read from the port, and if no tokens are available, the thread blocks until tokens become available. This is called *blocking reads*. Even though ideally communication channels should be unbounded, in practice, computers have limitations. With this scenario two types of deadlocks, real and artificial may arise. A *real deadlock* occurs when all processes block on a read, this is also the condition for termination. An *artificial deadlock* occurs when all processes are blocked and at least one is blocked on a write, due to the fact that the implementation operates with bounded buffers. To tackle write blocks, the buffer sizes are increased when needed.

PN models may also consume unbounded memory buffering tokens between actors. These models can be identified by a runtime policy that detects capacities that exceed a certain value. The application of this scheduling policy [56] executes non-terminating, bounded programs forever with bounded buffer sizes. Moreover, unbounded programs execute forever until the buffers reach a certain predefined size.

Feedback loops can be avoided either by using SampleDelay or creating enough tokens in the loop to break the data dependencies.

Ptolemy II's implementation supports mutations of graphs. To support adaptive behavior, a mutation mechanism is provided that supports addition, deletion, and changing of processes and channels, that could be non-deterministic in Kahn PN but it becomes deterministic in timed-PN. The Timed-PN domain extends PN with the notion of time. In this domain, time is global and is shared among all the processes. Processes can block themselves, waiting for time to advance and then resuming execution (delay). A priority event queue similar to the one used in DE is used to store the list of delayed processes. A *time deadlock* occurs when all processes are blocked, none on a write and at least one on a delay. Time advances only in those cases to the time when the first blocked process has to resume.

This MoC presents a great candidate for distribution since systems that obey Kahn's model are *deterministic.* Therefore they can be evaluated using a complete parallel or sequential schedule and every schedule in between, always yielding the same output results for a given input sequence. On the other hand, one of the key issues in PN is deadlock detection and resolution needed for termination and bounded execution. The implemented algorithms are centralized as well as the notion of time. These pose challenges when deploying a true distributed version. Keeping these mechanisms centralized with a distributed deployment introduces extra traffic and overhead when processes notify the director they are blocked.

3.4.5 COMMUNICATING SEQUENTIAL PROCESSES (CSP)

The model of computation used in the CSP domain is based on the CSP model first proposed by Hoare [57]. The communicating sequential processes domain, models a system as a network of sequential processes that communicate by passing messages *synchronously* through unidirectional broadcast channels. The transfer of messages between processes is via *rendezvous*, atomic, instantaneous actions that imply that both the sending and receiving of messages from a channel are blocking. If a process is ready to send a message, it blocks until the receiving processes are ready to accept it. Similarly if a process is ready to accept a message, it blocks until a sending process is ready to send it. Thus the communication between processes is rendezvous based as both the reading and writing processes block until the other side is ready to communicate. This model of computation is *non-deterministic* as a process can be blocked waiting to send or receive on any number of channels. It is also highly

concurrent due to the nature of the model. In this model of computation there is no shared state.

Some applications include specification and verification of concurrent aspects of system, research in concurrent theory and client-server databases and multitasking and multiplexing of hardware resources. Moreover, resource management and high level system modeling in embedded systems. It is implemented in various concurrent programming languages like Occam [58].

The domain implemented in Ptolemy II adds the notion of time and can be used with and without it. The notion of time promotes interoperability with other timed domains and to directly model temporal properties. Each actor is executed in a different thread. Since actors can request a delay, a process can block either when is trying to communicate with a process not ready to do so yet, or when it is *delayed* for some delta time, continuing when the director has advanced time by that length of time from the current time. Both cannot happen at the same time since it cannot block when it is delayed or delay when it is blocked. The director keeps track of the number of active, blocked and delayed processes to handle deadlocks. A real deadlock occurs when all the processes are blocked waiting to communicate, this is also the condition for the end of execution. A time deadlock occurs if at least one of the active processes is delayed. Time is global and controlled by the director. Actors can request a delay for a certain amount of time. Global time is updated whenever a time deadlock occurs (one of the processes is delayed).

A base class is provided for actors which use timing and non deterministic rendezvous in CSP.

Distribution issues of the current Ptolemy II implementation include global time for the timed version, a distributed synchronized message passing communication mechanism and deadlock detection for termination.

3.4.6 FINITE STATE MACHINES (FSM)

A finite state machine is a MoC that traditionally consist of a finite set of inputs, a finite set of outputs, a finite number of states (with a distinguished initial state), transitions between the states and actions defining the outputs. States store information about the past reflecting input changes. Transitions are labeled with input symbols. FSM are usually represented by a state-transition diagram but transition tables are also used. FSM is an intuitive way of capturing and representing a design that makes formal analysis and verification possible. FSMs are easily mapped to either hardware or software implementations thus can be applied to hardware/software co-design [59]. But FSM is not as expressive as the other computational models and inadequate for describing most complex systems where the number of states can become unbounded. On the other hand, combined FSM with other MoC can allow modeling of *hybrid systems* and modal models [60].

Finite state machines are widely used in different areas, and they are very well suited to model control logic for programs.

The FSM implementation is different from the other Ptolemy II domains. The component represent *state* instead of actors, the connections represent *transitions* between states. The visual rendering represents state-transition diagrams rather than dataflow. Execution is a strictly ordered sequence of state transitions.

Due to its sequential nature it is not an interesting option for distributed simulation.

Time is not represented in the current implementation. *Timed automata* [61] allows the specification of time. That could enable verification of temporal properties through model checking with tools like Uppaal [62].

3.4.7 CONTINUOUS TIME (CT)

The Continuous Time (CT) domain aims to help the design of systems that follow have continuous dynamics. Models in the CT domain have the form of ordinary differential equations (ODEs):

$$dx/dt = f(x, u, t)$$

$$y = g(x, u, t)$$

The first equation represents *system dynamic*s and the second th*e output of the system*. An *initial condition* of the system is also required. As for the variables, x is the state of the system, u is the input, and y is the output. Time t in the model is continuous, and dx/dt is the derivative of x with respective to time.

Some examples of applications are analog circuits and mechanical and aeronautical systems. As for embedded systems, it plays a key role to model the continuous physical systems with which embedded systems interact. Simulink, Saber, VHDL-AMS and Spice circuit simulators are alternative implementations.

The simulation of a continuous-time system requires solving the ODEs of the system. For that purpose, the director requires the help of one or more solvers. Their purpose is to find a fixed-point, a set of functions of time that satisfy all the relations. There is a variety of ODE solving methods developed over centuries. A class of them, called the time marching methods, discretizes time into discrete points and finds the value of x at these points at an increasing order of time. Ptolemy II implements some of the time marching methods, like the forward Euler method, the backward Euler method, the 2(3) order Runge-Kutta method, and the trapezoidal rule method.

The CT domain in Ptolemy II supports the interaction with event-based domains, like the discrete event (DE) domain and the finite state machine (FSM) domain, which yields the mixed-signal modeling and the hybrid system modeling. In order to interact with discrete domains, some actors in the CT domain are able to convert continuous waveforms to discrete events, and vice versa. An actor that has continuous input and discrete output is called an *event generator*; an actor that has discrete input and

continuous output is called a *waveform generator*. Event detectors are built in the CT domain library, so that the CT domain can always provide event-based interface to other domains when needed.

One distinct characterization of the CT model is the continuity of time. This implies that a continuous-time system have a behavior at any time instance. The simulation engine of the CT model should be able to compute the behavior of the system at any time point, although it may march discretely in time. In order to achieve an accurate simulation, time should be carefully discretized. The discretization of time, which appears as integration step sizes, may be determined by time points of interest (discontinuities), by the numerical error of integration, and by the convergence in solving algebraic equations.

Time is also global, which means that all components in the system share the same notion of time. Scheduling is static.

3.5 EXPERIMENTAL DOMAINS

Less matured domains are shortly described in the following:

3.5.1 GIOTTO

Giotto provides a programming abstraction for embedded control systems with hard real-time constraints, where periodic events dominate. Giotto defines a software architecture for the implementation which specifies its functionality and timing. Giotto is platform independent and can be compiled on different heterogeneous platforms. This is possible because it separates functionality and timing from mapping and scheduling. Giotto provides time predictability of the models. Resource allocation is pre-computed. Tasks can be programmed in any language.

The Giotto domain in Ptolemy II has actors representing the various tasks. The abstraction of hierarchy offers the actors the liberty of being models within themselves with any arbitrary model of computation. The concept of ports is similar in Giotto and Ptolemy, with input ports to the model representing sensors and output ports to the model representing actuators. The period of a Giotto iteration is specified using a parameter in the Giotto director, and the frequency of tasks are specified by parameters within the actors representing the tasks.

The Ptolemy II Giotto model can be compiled to a Giotto program and C code. The platform upon which it executes is a real-time flavor of Linux developed by Kansas University, KURT-Linux.

The Giotto domain is highly preliminary.

3.5.2 SYNCHRONOUS REACTIVE (SR)

The Synchronous Reactive (SR) domain models systems as components with both infinite processing resources and infinite communication bandwidth. An action of a

component is instantaneous just like data transfer between components [64]. This model of computation models the discrete steps in which the time advances instead of the continuous time. It has a global notion of time.

It is good at describing concurrent and complex control logic, safety-critical real-time applications due to its precise control of the timing of events (tight synchronization). Implemented in a number of (synchronous) languages as Esterel [69], Signal[68], Lustre [70] and Argos [71].

The synchrony hypothesis means that all events start and finish in the same instant. Actors respond to inputs from the environment, and the output events are synchronized with them since the components are infinitely fast. Moreover, it allows actors to execute multiple times in one instant. Therefore certain systems can be modeled under SR that would otherwise not be valid in other domains.

Execution of an SR system occurs at a sequence of global, discrete, instants called ticks. At each tick, each signal either has no event (is absent) or has an event (is present, possibly with a value). Signals are related by functions that have signals as arguments and define signals. Thus, at each tick, signals are defined by a set of simultaneous equations using these functions. In general, directed cycles are permitted.

The *components* represent relations between input and output values at each tick, are usually partial functions with certain technical restrictions to ensure determinacy. The *connections* between components represent data values that are aligned with global clock ticks. Unlike discrete-time models, a signal need not have a value at every clock tick. A solution is called a fixed point, and the task of a compiler is to generate code that will find such a fixed point.

A drawback of this MoC is that it over-specifies systems that don't need tight synchronization, which thus limits the implementation alternatives and makes distributed systems difficult to model.

3.5.3 DISCRETE TIME (DT)

The Discrete Time (DT) domain is a timed extension of the Synchronous Dataflow (SDF) domain. Like SDF, it has static scheduling of the dataflow graph model. Likewise, DT requires that the rates on the ports of all actors be known beforehand and fixed. DT handles feedback systems in the same way that SDF does, but with additional constraints on initial tokens.

Because of the inherent concurrency occurring within SDF dataflow graph models, there are two notions of time in DT, global time and local time. Global time increases steadily as execution progresses. Moreover, global time increments by fixed discrete chunks of time based on the value of the *period* parameter. On the other hand, local time applies to each of the actors in the model. All the actors have distinct local times as an iteration proceeds. The local time of an actor during an iteration depends on the

global time, period, firing count, port rates, and the schedule. These local times obey the following constraint:

Global Time <= Local Time <= (Global Time + period)

The DT domain is an experimental domain.

3.5.4 COMPONENT INTERACTION

The component interaction (CI) domain models systems that blend data-driven and demand-driven styles of computation. As an example, the interaction between a web server and a browser is mostly demand-driven. When the user clicks on a link in the browser, it pulls the corresponding page from the web server. A stock-quote service can use a data-driven style of computation. The server generates events when stock prices change. The data drive the clients to update their displayed information. Such push/pull interaction between a data producer and consumer is common in distributed systems, and has been included in middleware services, most notably in the CORBA event service. These services motivated the design of this domain to study the interaction models in distributed systems, such as stock-quote services, traffic or weather information systems. Other applications include database systems, file systems, and the Click modular router.

An actor in a CI model can be active, which means it possesses its own thread of execution. For example, an interrupt source of an embedded system can be modeled as an active source actor. Such a source generates events asynchronously with respect to the software execution on the embedded processor. CI models can be used to simulate and study how the embedded software handles asynchronous events, such as external interrupts and asynchronous I/O.

3.6 SUMMARY

Making distributed versions of a MoC poses a series of challenges.

Execution mechanisms: Schedule based mechanisms require a centralized coordination mechanism that dispatches the schedule, this creates overhead traffic. On the other hand, process based mechanisms show more autonomy of the different elements. One of the main problems in this approach is deadlock. There are two approaches, centralized or distributed. Several centralized approaches are implemented in the various domains in Ptolemy. Distributed deadlock detection mechanism are proposed in [72], [73]. Centralized approaches are notably simpler for detection and resolution than the distributed ones. In either case, significant traffic overhead has to be introduced. How that overhead compares with the overall execution depends on both the MoC and the systems. An interesting research path is to find deadlock patterns for different MoC and tune execution mechanisms to minimize them.

Communication mechanisms: Both synchronous and asynchronous message passing is supported for distributed environments. For Java implementations, Java RMI offers a good solution, keeping the semantics of method calls over a network. RMI is as RPC in general a rendezvous, so they are inherently synchronous.

Time: Distributing time is an active area of research that presents a series of problems. There are two main approaches to tackle distribution of time, conservative and optimistic. Conservative approaches avoid processing events out of order (violating the causality constrain) whereas optimistic detect events out of order at runtime and implement mechanisms to recover from them. The DDE domain implements a conservative approach that uses null messages to avoid deadlocks. Other conservative methods include synchronous execution using barriers. The most obvious drawback is that conservative approaches cannot fully exploit the inherent concurrency of the application. Time warp is a well known optimistic approach where modifications due to events processed out of order are undone (rollback). Rollback can lead to very inefficient use of computation and communication but can be alleviated with advanced mechanisms. Incremental checkpointing techniques have been recently implemented in Ptolemy II that can be applied in a time warp approach [74]. Neither of the approaches has proved to be applicable for the generality of systems since their efficiency is dependent on the problem and implementation. A general method that is optimal for most applications may not exist.

Among the different MoC we choose to implement SDF for ADS. SDF can be scheduled statically what allows to apply divide and conquer in tackling the problem. The next chapter is devoted to scheduling where were solve the problem of finding an optimal parallel schedule that will expose the parallelism of the model. The communication mechanism is dealt with in Chapter 5, where the implementation of a distributed message passing mechanism is described. Different issues arise as identification of receivers in a distributed deployment. Distributed time is a complex ongoing research issue that requires a distributed infrastructure that is yet to be developed. We leave distributed time for future work, where we give some pointers on how add it to ADS. Fortunately, SDF does not include the notion of time. Moreover SDF is a popular formalism that is suited for a large number of applications and thus it is worthwhile to develop an efficient parallel implementation.

CHAPTER 4

SCHEDULING

The key is not to prioritize what's in your schedule, but to schedule your priorities.
Stephen R. Covey

This chapter introduces the scheduling problem. Scheduling is a complex problem in its general form that is still an active ongoing research field. Certain constraints have to be assumed in order to find optimal solutions in an efficient way. These assumptions are described. Moreover, a survey of previous scheduling work is presented, with emphasis on parallel scheduling within the Ptolemy project. Scheduling has been one of the hot topics in the Ptolemy project, allowing different schedulers to be mixed within the same system.

Scheduling is used in solving problems both inside and outside the computer systems world. It can be applied to a diverse range of problems as for example, tackling a to-do list of everyday life duties, assigning tasks to a team of people or optimizing assembly lines. But most relevant to this work, scheduling is a key part of the execution mechanisms in various MoC like SDF.

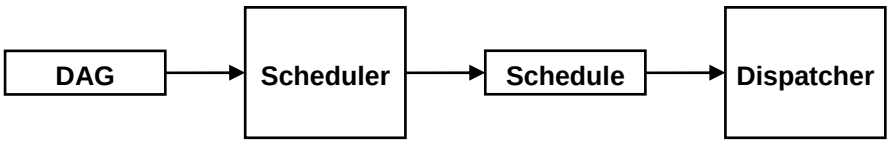


FIGURE 4: BUILDING BLOCKS OF THE SCHEDULING MECHANISM

The objective of this chapter is to propose a scheduling policy that minimizes the execution time of a simulation by using a distributed platform given a MoC. As shown in Figure 4, the input for the scheduler is a DAG that represents the dependencies of the actors in a model, annotated with input and output rates . The output is a schedule that is used by the dispatcher (in our case the director of the domain) to coordinate the execution of the simulation. The MoC and the topology of the model define the constraints that must be satisfied by the schedule and are represented in the DAG. The current implementation of the SDF Scheduler produces a sequential schedule since it is executed on one processor. Distributed platforms enable to make a better effort to

exploit parallelism of the SDF models, both the inherent exposed by the topology and the iterative nature of the models.

4.1 THE SCHEDULING PROBLEM

Generally speaking, scheduling a set of tasks is to determine their *execution order*; that is, when to execute which task. In the case of a parallel/distributed system, it is also required the *assignment of tasks* to processors.

The *goal* of scheduling is typically to *maximize throughput* and/or *minimize execution time* also called *makespan*. Scheduling becomes more complex for real-time systems [75], where the primary goal is to *meet the deadlines for every task*.

A distinction should be made between *local* and *global* scheduling. A local scheduling discipline determines how the processes resident on a single processor are allocated and executed; a global scheduling policy uses multiple processors to exploit parallelism. Information about the system can be used to allocate tasks to multiple processors with the view of optimizing a performance objective. Ptolemy implements a local scheduling policy that makes no effort to exploit parallelism and it is executed in a single processor. Our objective is to devise a global scheduling policy which preserving the constraints of the model, minimizes the makespan of the execution. Two different types of constraints appear in SDF models, *precedence* and *production/consumption rates*. Both have to be preserved when scheduling in order to assure correctness.

4.2 REPRESENTATION

Directed graphs [76] are probably the most popular representation of precedence constraints that correspond to dependences among tasks. The nodes in these graphs represent independent operations or parts of a single program which are related to each other in execution order.

A scheduling graph can be defined by its vertices V and its edges E as $G = (V, E)$, the vertices are tasks $(n \in V)$, while the directed edges $(E \subseteq V \times V)$ give precedence among the tasks; the edges are denoted by (n_i, n_j). A Directed Acyclic Graph (DAG) is a directed graph with no cycles; that is, for any node n, there is not path that starts and ends on n. Nodes and edges can have weights associated. The weight of a node is called the *computation cost* and is denoted by $w(n_i)$. The weight of an edge is called the *communication cost* of the edge and is denoted by $c(n_i, n_j)$. The source node of an edge is called the *parent node* while the sink is called node is called a *child node*. A node with no parent is called an *entry node* and a node with no child is called an *exit node*. The communication-to-computation-ratio (CCR) of a parallel program is defined as its average edge weight divided by its average node weight. Since there is no notion of time in SDF, weights are not used, but arcs are annotated with production/consumption rates. Figure 5 shows an SDF model that shows dependencies

among the actors. The production/consumption rates of the ports are not shown in Ptolemy II's user interface (vergil), but have been made explicit here. This information is relevant for a correct computation of the schedule and gives SDF the properties mentioned in the previous chapter.

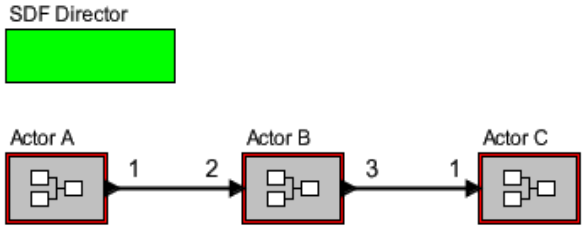


FIGURE 5: ANNOTATED GRAPH IN PTOLEMY II, PRECEDENCES AND RATES ARE SHOWN

The `ptolemy.kernel` package implements the infrastructure for clustered graphs that provide the abstract syntax of models. Moreover, the `ptolemy.graph` package provides support for mathematical graphs and algorithms to manipulate and analyze them that can be used in different applications. Mathematical graphs are simpler than Ptolemy II clustered graphs in that there is no hierarchy, and arcs link exactly two nodes. Both undirected and directed graphs are supported as well as DAGs and weights.

4.3 SCHEDULING OF DEPENDENT TASKS

The topology of models in Ptolemy represents precedence constraints among the actors that dictate their processing order. These constraints must be preserved by the schedule to ensure correctness of the execution. A precedence constraint between two actors τi and τj, denoted by $\tau i \rightarrow \tau j$, exists if the execution of τi precedes that of τj. In other words, τj must await the completion of τi before beginning its own execution. Satisfying precedence throughout execution guarantees correctness. Precedence constraints are relationships that can be described through a graph where the nodes represent tasks and the arrows express the precedence constraint between two nodes. Any valid SDF model can be translated into a DAG otherwise it cannot be scheduled.

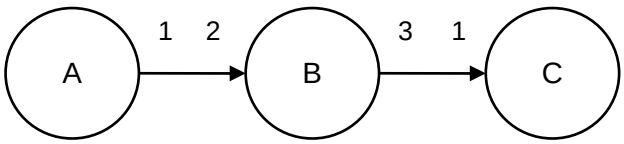


FIGURE 6: DAG DERIVED FROM THE SDF MODEL IN FIGURE 5

If τi is connected by a path to τj in the precedence graph then $\tau i \rightarrow \tau j$. A general problem concerns tasks related by complex precedence relationships where n successive instances of a task can precede one instance of another task, or one instance of a task precedes m instances of another task. This can be applied to clustering.

The `ptolemy.graph.sched` supports the construction of schedules. The implementation of the scheduling policy for the SDF domain can be found under

`ptolemy.domains.sdf.kernel.SDFScheduler`. It computes a sequential schedule that does not expose the parallelism of the topology. Therefore a new scheduler that can compute a *parallel* schedule is required. By parallel we mean a schedule that can expose the parallelism of the topology. At the same time satisfy the precedence constraints have to be satisfied for correctness as well as firings must match the production and consumption rates to execute in bounded memory.

4.4 THE MULTIPROCESSOR MODEL

A multiprocessor model is a model of parallel computation based on a set of communicating sequential processors PEs representing the platform where the simulation (in our case) is computed. This platform is assumed to be a network of processing elements (PEs). Each PE is composed of a processor and a local memory unit so that the PEs do not share memory and communication relies only on message passing. Tasks have to be assigned to the processors. PEs are *homogeneous* when all processors have equal task processing speeds or *heterogeneous* when the processors differ in their speeds. The PEs are connected by a network with a certain *topology*. The topology may be fully connected or of a particular structure such as a *hypercube* or *mesh* [77]. Some recommendations for scheduling in multiprocessor environments are given in [78].

Our intended target multiprocessor platform is a tightly coupled parallel architecture, a cluster of computers. We assume *homogeneous* PEs in our problem definition and a *fully connected network topology*. It is assumed that the actors in the model can be executed on any processor. The speed of each processor is constant and does not depend on the actor.

4.5 NP-COMPLETENESS

The scheduling problem on a single or multiprocessor system is in general a NP-Complete problem [79]. NP-Complete means that it is at least as difficult to compute as the hardest problem in the family NP, which is the family of problems capable of being solved by non-deterministic algorithms in polynomial time. A NP-complete problem can be solved in polynomial time when a bounded non-deterministic choice is allowed to guide the algorithm towards a solution. x

Consequently, many heuristics have been developed to generate adequate (but possibly sub-optimal) schedules [80]. Heuristics produce an answer in less than exponential time but does not guarantee an optimal solution. There are few known polynomial-time scheduling algorithms even when severe restrictions are placed on the task graph representing the program and the parallel processor model. Common simplifying assumptions include *zero inter-task communication times, full connectivity of PEs* and *availability of unlimited number of PEs* among others. We adopt these assumptions to allow for a polynomial time algorithm.

4.6 STATIC VS DYNAMIC SCHEDULING ALGORITHMS

Depending on whether the characteristics of the processing load (processing times, communication, data dependencies, synchronization requirements) are known a priori, the scheduling problem can be *static* or *dynamic* [81].

4.6.1 STATIC SCHEDULING

Static scheduling is possible when all the relevant information for scheduling is available before runtime. Static scheduling involves assigning the tasks to processors and then not allowing them to move. This minimizes overhead and turnaround time for the user but is not responsive to changes in the execution environment. One of the major benefits of the static model is that it is easier to implement from a deployment view. The placement of tasks is fixed *a priori*; it is easy to monitor the progress of the computation and hence termination of processing is simplified. Using the static model, each host is told from where to get the input parameters that are needed, or where to send the outputs of the programs.

Unfortunately, since it is often assumed that nothing changes in the system's environment, the static model does not allow for the very real possibility that one of the nodes selected to perform a service fails or becomes isolated due to network failure. It is *fault intolerant*.

There exist different static scheduling methods. Since static scheduling is a search problem the simplest method is *brute force* where no information is used to guide the search with the obvious lack of efficiency. Other state space search methods utilize heuristics like *simulated annealing* [82] and *genetic algorithms* [83]. Simulated annealing has a higher risk of getting stuck in a local optimum therefore not delivering the optimal solution. An example of a parallel genetic algorithm used for scheduling is found in [84].

Mathematical programming methods to resolve task scheduling problems typically consist of a definition of:

- *An objective function:* to be minimized, say the program execution time,
- *Set of constraints*: to preserve properties, such as task precedence relations,
- *The application of a method*: for solving complex constrained optimization problems, such as dynamic programming or heuristic techniques.

All these methods are time consuming and computation intensive and not guaranteed optimal. A taxonomy of static parallel scheduling algorithms is given in [85]. The constraints imposed by the SDF model of computation allow for static scheduling.

4.6.2 DYNAMIC SCHEDULING

In dynamic scheduling, only a few assumptions about the parallel program can be made before execution, so scheduling decisions have to be made on-the-fly. The goal of a dynamic scheduling algorithm includes not only the minimization of the program completion time, but also the minimization of the scheduling overhead; the cost of running the scheduler. The *load distributing strategy* determines how programs will be placed on remote machines. It uses an information policy to determine which information is to be collected from each machine in the processor pool, at what frequency, and also how often the local information should be exchanged with other machines.

In a strictly dynamic scheduling model, the tasks that comprise a parallel or distributed job are assigned to processors based on whether a processor predicts that it can provide an adequate quality of service to a task. The meaning of quality of service is dependent on the application. For example: whether a maximum bound can be placed on the time a job will have to wait before starting execution; the minimum time that a given job will be able to execute without interruption; and, the relative speed of a processor when compared to others in the processing pool. If the processor is assigned too many tasks, it may invoke a *transfer policy* to decide *whether* to transfer some tasks, and *which* tasks to transfer.

A *location policy* determines which processor(s) will receive the tasks. There are two important models for location policies: sender-initiated and receiver-initiated. These depend on whether the sender polls potential targets for task transfer, or whether processors that are willing to receive extra tasks advertise the fact, respectively.

The advantage of dynamic scheduling over static scheduling is that the system needs not to be aware of the run-time behavior of the application before execution. Process-based MoCs and well as DE require dynamic scheduling or coordination.

4.7 GRANULARITY, PARTITIONING AND CLUSTERING

Ptolemy models are the result of a *partition* of functionalities into actors. How the partitioning of the system modeled is done, has an impact that may limit or broaden options for optimization of the scheduling. Thus, partitioning strongly affects scheduling and can for example alleviate the overheads introduced by communication and synchronization. Efficient distributed processing is heavily influenced by the way the partition is made and it consists in finding a trade-off between the number of processors to use, number of tasks to execute and amount of overhead introduced. Finding the right actor *granularity* can achieve such a trade-off. It is defined as the ratio between task computation time T and task communication overhead C. Fine-grained tasks correspond to a small (T/C) and coarse-grain tasks to a high (T/C). Since there is a direct correlation between program efficiency and granularity, a technique

for optimization is to *cluster* a set of fine-grain tasks into coarser-grain partitions. Clustering, is important when it involves a collection of actors that exchange a large amount of data. The goal is to eliminate the overhead caused by inter-task communication while maintaining a high degree of parallelism. Several heuristics have been suggested for clustering [86]. There are mainly 3 generic classes:

Critical Path partitioning: Since fine-grain tasks relying on the critical path of the program have to be executed sequentially anyway, those nodes can be identified on such a path and cluster them into a partition. This method is called Vertical Layered Partitioning [87]. The algorithm clusters the dominant sequence of a directed acyclic graph (DAG), which is the critical path of a DAG whose nodes have been allocated to processors. The critical path of a DAG is the path connecting the programs which dominate the execution time of the DAG. If the programs on the critical path are not optimally scheduled, the resulting execution time for the job will be non-minimal.

Communication Delay elimination partitioning: Here the communication overhead is used as criteria for clustering tasks into partitions. The approach is to cluster the node as well as its successors, provided the communication overhead is not prolonged. The advantage of this scheme is that it provides an upper bound on the number of allocated processors.

Task Duplication partitioning: Task duplication allows one to eliminate communication costs by duplicating the tasks on the PEs. This not only eliminates communication costs but also preserves the original program parallelism. The disadvantages are increased space requirements and increased synchronization overheads

Since communication costs are not the main concern we assume LGDF (Large Grain Data Flow) where the CCR becomes small.

4.8 SCHEDULING IN PTOLEMY AND PREDECESSORS

Most of the scheduling work within the Ptolemy project has been carried out in the earlier versions of the tool. Starting from Gabriel, the scheduler is perhaps the most sophisticated part of the tool [88]. Gabriel's scheduler receives two inputs, an acyclic precedence expansion graph (APEG) and a target architecture specification. The APEG is derived automatically from a block diagram and is an acyclic graph in which each node represents the invocation of a star. The target architecture provides the processor topology and the IPC costs. The IPC costs as well as the execution times for the actors (stars in Gabriel) are estimates given by the star designer(s). The main contribution of [89] is to extend SDF to a multidimensional model in order to exploit data parallelism. It is not a general method and scheduling is not discussed. Information about the internal structure of the schedulers in Ptolemy Classic is given in [24]. The different functionalities implemented in the different methods are outlined. A description of the functionalities implemented in the schedulers in Ptolemy

Classic is given in [90]. Users can write their own schedulers and extend them from the existing ones. For single-processor schedules two basic scheduling techniques are described. The first one includes calculation of the firing rates and after choosing randomly runnable actors to be fired. A runnable actor is one with all inputs available in its input arcs. The second approach called *loop scheduling*, results in hierarchical clustering. Ptolemy Classic implements Hu's level based list, Sih's dynamic level and Sih's declustering scheduling techniques to map SDF graphs onto multiple-processors. The target architecture is provided to the scheduler including necessary information about the topology and intercommunication costs. Sih's dynamic level scheduling is described in [91] and applies dynamic level scheduling to heterogeneous processor networks. It accounts for inter-processor communication (IPC). Sih's declustering is described in [92]. Declustering exposes graph parallelism in order of importance comparing the trade-off between parallelism exploitation and IPC cost.

In [93], the problem of determining the optimal blocking factor for SDF graphs is addressed. It contributes to the theoretical understanding of the dynamics of increasing the blocking factor. It does not report on the construction of rate-optimal blocked schedules for multiprocessors stating that it is still a difficult problem.

A clustering algorithm for optimizing parallel schedules is reported in [94]. The result of clustering exposes extensive parallelism for the multiprocessor scheduler. The internals of the clusters are scheduled with single processor SDF schedulers.

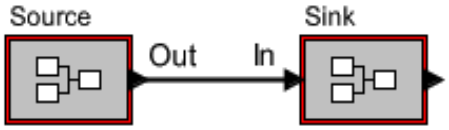
4.9 SCHEDULING FOR THE SDF DOMAIN

Model in the SDF domain provide all the information required for scheduling prior to runtime what allows for static scheduling. Constraints in the form of precedences among actors and production and consumption rates have to be preserved in order to ensure correctness of the execution. Consistent and immutable production and consumption rates allow for execution in bounded memory. The rates of ports are specified using parameters on each port named `tokenConsumptionRate`, `tokenProductionRate`.

The existing implementation of the SDF scheduler adds little overhead and results in execution without deadlock. Here we describe how the existing scheduling mechanism works. This scheduler provides loop-free, sequential schedules suitable for use on a single processor. No attempt is made to optimize the schedule by minimizing data buffer sizes, minimizing the size of the schedule, or detecting parallelism to allow execution on multiple processors.

The construction of the schedule in SDF consists in two different phases. First, *solving the balance equations* [95] where the minimal integer number of repetitions of each actor is computed. That is called an iteration of the model, which results in the queues containing the same amount of tokens as prior to the iteration. If no solution exists, the models cannot be executed in bounded memory. Afterwards, *sorting* of the

actors according to their dependencies is performed, they can be fired whenever their input ports contain enough tokens, for example, source actors (without input ports) satisfy this condition before any other actor has been fired.



For every connection between ports in a model an equation of the following form can be defined:

*Firings(Source) * ProductionRate(Out) = Firings(Sink) * ConsumptionRate(In)*

These equations relate the number of tokens created and consumed during an iteration. For the example from Figure 5 we obtain A = 2B and 3B = C. The minimal integer solution to them is given by the *firings vector* (2, 1, 3). Therefore 2 iterations of A, one of B and 3 of C bring the queues back to their original empty state. Usually balance equations have an infinite number of linearly dependent solutions. In some cases, no solution exists; such graphs are called *inconsistent* and result in either deadlock or unbounded memory usage. Inconsistent graphs can still be executed using the PN domain, if necessary.

After the firing vector is computed; ensuring dependences are preserved (producers fire before consumers) yields correctness of the execution. This is done by simulating the production and consumption of tokens. This is done in a *depth first* fashion which optimizes memory consumption by minimizing the number of tokens in transit.

In order to perform this simulation, at least one actor that is ready to fire has to exist. Actors are ready to fire when they have no unfulfilled input ports, for example source actors that contain no input ports. In some cases where the model contains cycles, such an actor does not exist even though a valid firing vector exists. Then, no actor can fire, since they all depend on the output of another actor. This situation is known as *deadlock* and it can be prevented by creating tokens during initialization. This can be done in 2 different ways, by inserting delay actors or declaring initial tokens can be sent from any port in the `initProduction` property.

The vectorization factor parameter controls the granularity of the schedule, thus it can be used to cluster communication. The vectorization factor is multiplied by the number of times than an actor would fire in the smallest valid schedule.

4.10 PARALLEL SCHEDULING FOR THE SDF DOMAIN

If a workable schedule for a single processor can be generated, then a workable schedule for a multiprocessor system can also be generated. Trivially, all the computation could be scheduled on one of the processors. But in general, the makespan can be reduced substantially by distributing the load more evenly. A source of parallelism in Ptolemy II SDF models are actors that are at the same level after a

topological sort of the associated DAG, therefore they can be fired in parallel. This is called *inherent parallelism* of the model.

What is needed in the first place in order to exploit this parallelism is to generate a schedule of the model that exposes such parallelism. This requires analysis of the DAG and can be achieved by performing a topological sort. The topological sort produces an ordering in levels that represent the actors that can be executed in parallel. Due to the lack of knowledge about the run times for different actors, no better effort than a topological sort of the graph can be made. This results in linear complexity.

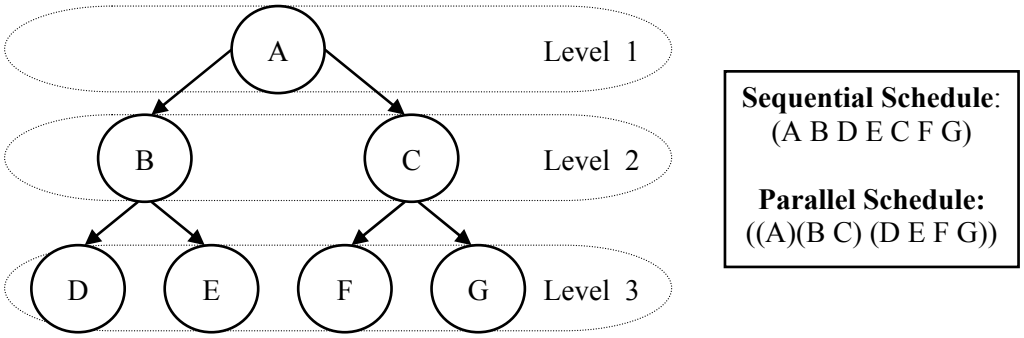


FIGURE 7: SEQUENTIAL AND PARALLEL SCHEDULES FOR A DAG AND LEVELS

In Figure 7 an example DAG is shown with the corresponding depth first sequential schedule and a breadth first parallel schedule. The parallel schedule is constructed as a sequence of the actors arranged by levels. The computation of the firing vector remains the same as in the existing `SDFScheduler`. In fact, the implementation of the parallel scheduler (`DistributedSDFScheduler`) extends the existing one, reusing much of the existing code. The existing infrastructure for creating schedules, provided in the package `ptolemy.actor.sched` supports parallel schedules. The `Schedule` consists of a list of schedule elements and the number of times they should repeat. A `ScheduleElement` can be a `Firing` class which corresponds to the number of firings of a single actor or an entire sub-schedule represented by a `Schedule`. The schedule that we compute contains a list of schedules that represent each level, each of them containing firings of actors of the form shown in Figure 7 and the structure in Figure 8.

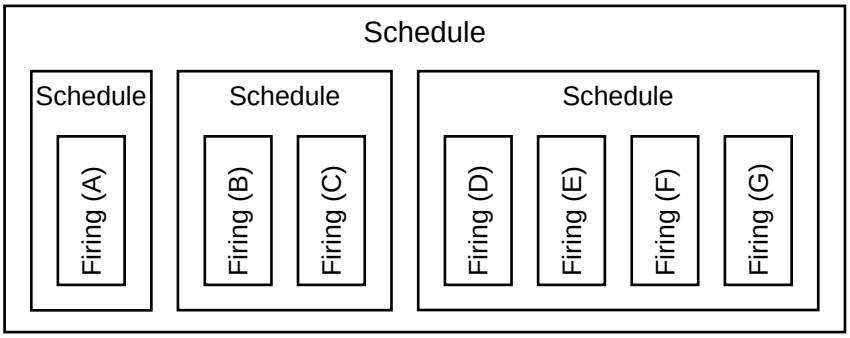


FIGURE 8: PARALLEL SCHEDULE STRUCTURE FOR FIGURE 7.

Correctness of the schedule is ensured since the solution of the balance equations and calculation of the firing vector reuses the exact same procedure and the sorting of actors preserves dependencies among them.

4.11 PIPELINING

Due to the iterative nature of the SDF MoC, the generated schedule implemented is for a single processor is called a Periodic Admissible Sequential Schedule (PASS) since it can be iterated over and over again. The generated parallel schedule is a Periodic Admissible Parallel Schedule (PAPS) [95]. The schedule should be periodic because of the assumption that we are repetitively applying the same program on an infinite stream of data. It is admissible because the blocks will be scheduled to run only when data are available. It is parallel because more than one processing resource can be used. This periodicity exposes another source of parallelism that can be exploited applying pipelining for the simulation. Pipelining has been applied in the design of processors and digital electronics to increase performance where pipelining reduces cycle time of a processor and hence increases instruction throughput, the number of instructions that can be executed in a unit of time. In our case, we use pipelining to alleviate the effect of the dependencies.

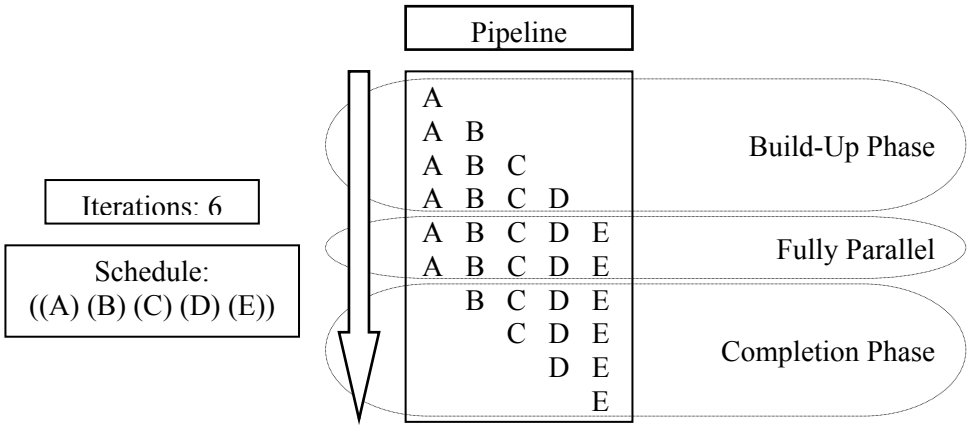


FIGURE 9: PIPELINING.

Since a given actor can be fired whenever their input requirements are fulfilled, actors do not have to stall until a complete execution of the iteration is performed. This is illustrated in Figure 9. In this example, a parallel schedule of a sequential topology is executed in a pipelined manner. Without pipelined execution, sequential topologies cannot benefit from distributed simulation. As shown, a fully parallel execution of the model can be achieved if the number of iterations is sufficient. Pipelining execution consists of a build-up phase where tokens are generated to allow all actors to fulfill their inputs in a way the precedences are preserved. This is followed by fully parallel phase where most speedup is achieved. Finally a completion phase takes care of processing the remaining tokens still in transit. This has to be performed in a way such that the number of iterations of the simulation is preserved.

Tokens in transit denote a higher overall memory consumption but since distributed simulation partitions the memory consumption it becomes insignificant.

Correctness is ensured since for any given actor the input tokens order is preserved. There is an exception, and that is models that contain cycles. Since cycles are broken by including delayed tokens, it is important to notice that such models cannot benefit from the pipelined execution in general. Even though a sufficient number of delayed tokens can break the cycle and allow execution, this means the designer has to take this into consideration and this is against the objectives of this work.

Further details concerning the development and implementation of the scheduling and dispatching mechanisms, including pipelining are included in the next chapter.

4.12 OPTIMALITY

The proposed schedules are optimal since they result in makespans that are the sum of the actors in the critical path. If an scheduler yields smaller times, it means that two (or more) elements in the critical path are placed at the same level in the schedule, thus violating precedence constraints.

4.13 SUMMARY

The scheduling problem has been introduced in this chapter. Directed graphs are a popular representation of the scheduling problem when considering dependent tasks. Different multiprocessor models are described is our intended platform for distributed simulation. Certain common simplifying assumptions including *zero inter-task communication times, full connectivity of PEs* and *availability of unlimited number of PEs* are adopted to allow for a polynomial time algorithm. The two main categories of scheduling are introduced (static and dynamic). Our target problem is of the static category meaning that schedules can be pre-computed prior to simulation. Granularity, partitioning and clustering are introduce and how they affect the scheduling problem. A survey of prior scheduling techniques implemented in the Ptolemy project is presented. Details about the current SDF scheduling policy are described. A parallel schedule computation technique is proposed that includes pipelining techniques for dispatching. This technique is optimal since it exploits both inherent parallelism of the model and the one arising from the iterative nature of SDF.

In the next chapter we proceed with the implementation of ADS in Ptolemy II.

CHAPTER 5

ADS IMPLEMENTATION IN PTOLEMY II

Programming today is a race
between software engineers striving
to build bigger and better idiot-proof
programs, and the Universe trying
to produce bigger and better idiots.
So far, the Universe is winning.
Rich Cook

Our implementation of ADS in Ptolemy II is described in this chapter. An implementation enables experiments and gathering of empirical results to draw conclusions in the next chapter. Furthermore, the implementation described here is included in the latest release of Ptolemy II, thus ADS technology is now available to the ES modeling and design community.

In the original SDF implementation, a given Ptolemy model is executed in the local machine. The execution takes place in a sequential manner. First a sequential schedule is computed then executed. This is very inefficient if we consider models with inherent parallelism. The iterative nature of the SDF computational model also allows for improvement when applying pipelining techniques. Distributing the execution of Ptolemy models enables for real parallel execution and bigger models. But distributing execution in an efficient manner requires expertise and time and it is highly error prone.

In the following we give an overview of the overall approach for distributed simulation. The approach is introduced and the different elements and roles are enumerated and described. Then ADS is introduced along with the technologies required for realizing it. Our implementation extends Ptolemy II's architecture, which provides the infrastructure for actor based simulation. Therefore the preexisting architecture is described to help the reader understand the remainder of the chapter. The different elements that form ADS are described in further detail in connection with the architecture. An architecture poses a number of challenges when moving from a single computer to a distributed world that are described and resolved.

5.1 OVERVIEW

The implementation documented in this chapter extends the existing classes that implement SDF and the underlying infrastructure. In order to enable distributed execution of the model, a distributed platform is required. Our distributed platform consists of a set to processing nodes. We provide server software that enables computers for distributed simulation. More than one server can be set up on a single computer. Finally a peer discovery mechanism allows servers to register and provide their services thus facilitating finding such services for those demanding them.

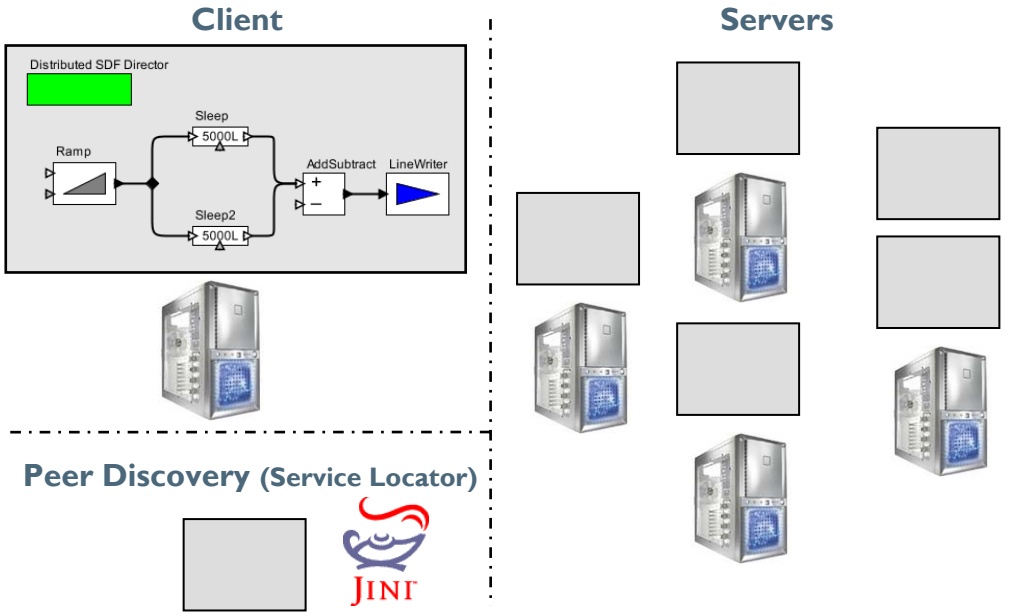


FIGURE 10: ADS OVERVIEW, CLIENT / SERVER APPROACH

Figure 10 shows a network of computers running a distributed platform with all the elements (the client with the model, different servers and the service locator). The service locator can be hosted anywhere in the network, either the client, any of the servers or a dedicated machine. A Service Locator has to be running in the network before the servers are launched for them to register the services they provide. Before the simulation is started, the server software has to be running on the servers. Servers stay dormant until the services they provide are requested. Once the distributed platform (Service Locator + Servers) is up and running, ADS is possible from the client.

From the user's point of view, using ADS only requires replacing the existing SDF Director by the Distributed SDF Director we provide.

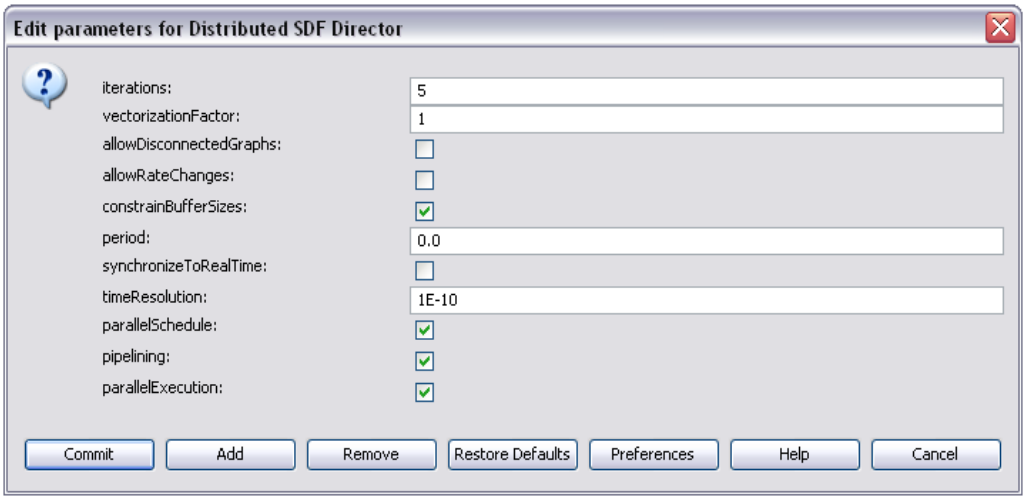


FIGURE 11: CONFIGURATION OPTIONS FOR THE DISTRIBUTED SDF DIRECTOR.

Figure 11 shows the configuration parameters for the Distributed SDF Director. Parallel schedule, pipelining and parallel execution have been added to enable control over the distributed execution. These parameters are elaborated later in this chapter.

Once the simulation is launched the following steps are performed:

1. Calculation of the parallel schedule.
2. Collecting the services.
3. Mapping the actors onto servers
4. Distributing actors onto servers
5. Connect distributed actors
6. Parallel dispatching of the schedule

If there are enough services, the different actors realizing a model are transparently distributed to different machines in order to perform a truly parallel execution. Transparency is provided by means of discovery and allocation of resources realized by the implementation. Then the distributed actors are connected over the network in a way that conforms to the topology described in the model. Finally the centralized dispatcher orchestrates the execution exploiting parallelism at the same time it ensures correctness of the simulation. The different elements that make this possible and glue it all together are further described in the following.

5.2 THE EXISTING PTOLEMY II SOFTWARE ARCHITECTURE

For a better understanding of the implementation details, we describe the existing software architecture provided by Ptolemy II software. The distributed SDF packages that implement ADS build on top of the existing Ptolemy II architecture. Here we present the architectural features that we consider most relevant and related to this work. Further information concerning Ptolemy II's software can be found in [40].

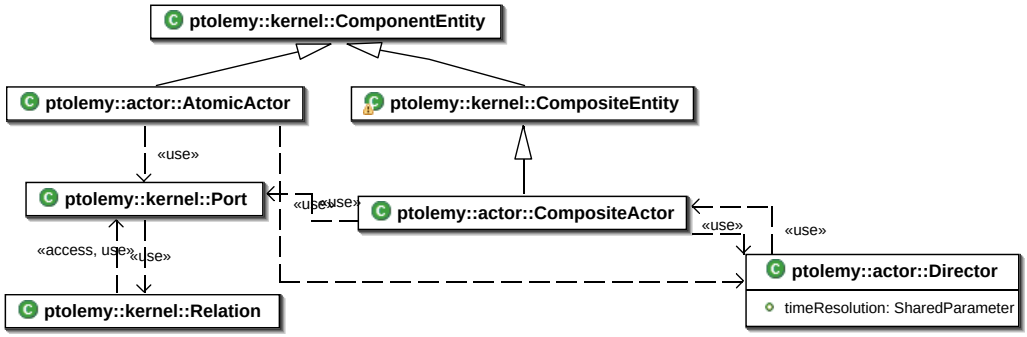


FIGURE 12: PTOLEMY MODEL CLASSES, DIRECTOR, ACTOR AND TOPOLOGY.

As previously said, Ptolemy II models aggregates components connected in a certain topology and a director that implements a MoC. A Ptolemy II model consists of a `CompositeActor` at the top level of the hierarchy. A `CompositeActor` is an aggregation of actors. It will normally contain a local `Director`. The topology is represented by hierarchical clustered graphs. Hierarchy and clustering is provided by `CompositeActor` and graphs are created though links between relations and actor's ports. Figure 12 shows the generic `Director` and `Actor` classes from the actor package and the classes implementing the topology. `Kernel` classes define the abstract syntax of models.

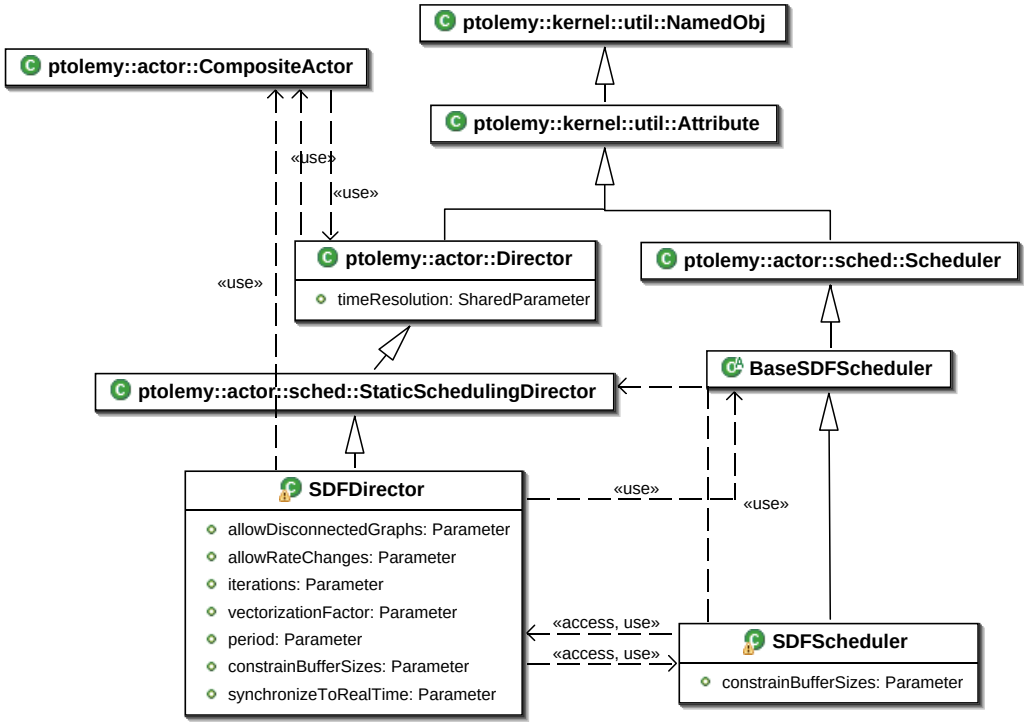


FIGURE 13: SDF.KERNEL PACKAGE, SCHEDULER AND DIRECTOR CLASSES HIERARCHY.

Different domains are organized in packages under `ptolemy.domains`. Domain packages are divided into subpackages that contain for example domain specific actors under `lib` or core classes under `kernel`. The `ptolemy.domains.sdf.kernel` package contains the main classes implementing the SDF MoC. Figure 13 shows the class hierarchy for the `SDFScheduler`, which extends a `BaseSDFScheduler` that provides basic scheduling features. Schedulers dictate the execution order of the

actors contained by a `CompositeActor`. Schedules are often cached to reuse for optimization as long as they are valid. Directors implementing domains that perform static scheduling, like SDF, use schedulers in the form of attributes to compute scheduling. `StaticSchedulingDirector` is the class supporting this; it does not implement a scheduling algorithm but defers it to its contained scheduler. `SDFDirector` extends `StaticSchedulingDirector` to take advantage of the infrastructure for supporting static schedulers. A `Director` is an `Attribute` as well, in this case of a `CompositeActor`. Attributes are `NamedObj`, which is a base class for almost all Ptolemy II objects which supports the name scheme.

The `Actor` interface exposes the available functionality of a component to the outside world in a standardized way common to all actors. It extends the `Executable` interface defining the action methods, which determine how to interact with an actor. Actors and directors should implement it. When running a simulation, the `preinitialize()` and `initialize()` methods have to be invoked exactly once in that order, followed by any number of iterations, followed by exactly one invocation of the `wrapup()` method. Figure 14 shows the methods declared by the `Executable` and `Actor` interfaces. An iteration is defined to be one invocation of the `prefire()` method followed by any number of invocations of the `fire()` method, followed by one invocation of the `postfire()` method. The `prefire()` method returns true to indicate that firing can occur. The `postfire()` method returns false if no further firings should occur. The `initialize()`, `fire()` and `postfire()` methods may produce output data.

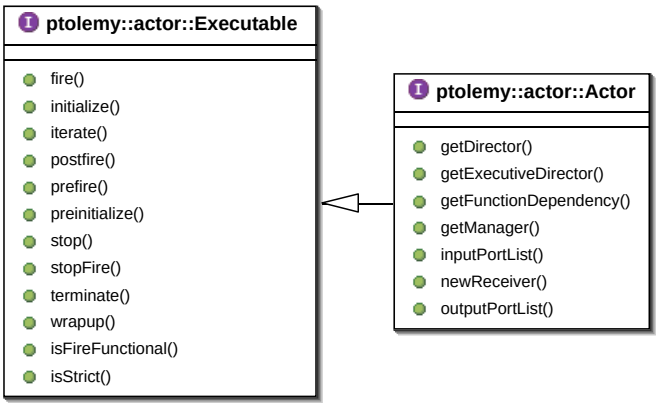


FIGURE 14: EXECUTABLE AND ACTOR INTERFACES.

`CompositeActors` as well as `AtomicActors` implement the `Actor` interface. An `AtomicActor` is an executable entity that does not contain other `Actors`. A `TypedActor` is an actor whose ports have types. Figure 15 shows the class hierarchy for the `Actor` interface, composite actor and atomic actor and how they are related with the scheduler and director of SDF.

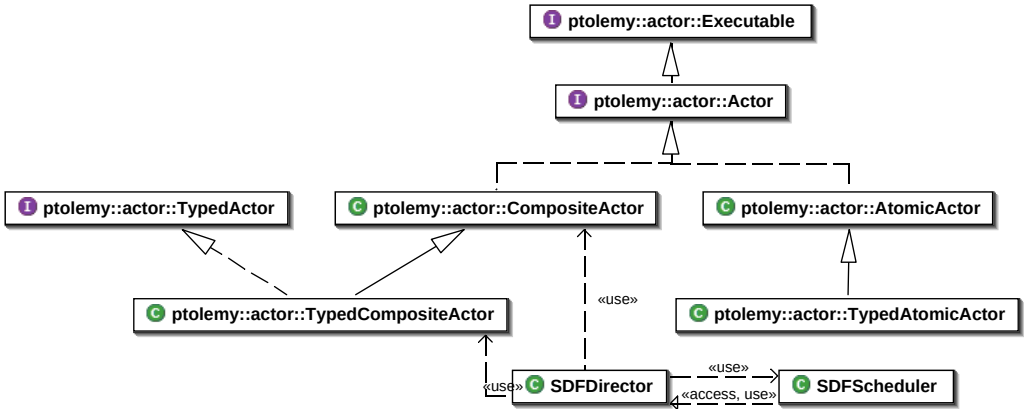


FIGURE 15: EXECUTABLE AND ACTOR INTERFACES HIERARCHY.

Both atomic and composite actors are entities. An `Entity` is a vertex in a generalized graph. Entities are intended for flat graphs. Derived classes support hierarchy (clustered graphs) by defining entities that aggregate other entities. A `ComponentEntity` is a component in a `CompositeEntity`. It might itself be composite, but in this base class it is assumed to be atomic (meaning it cannot contain components). Derived classes may further constrain the container to be a subclass of `CompositeEntity`. A `CompositeEntity` is a cluster in a clustered graph (meaning it is a non-atomic entity, in that it can contain other entities and relations). Figure 16 shows the entity class hierarchy.

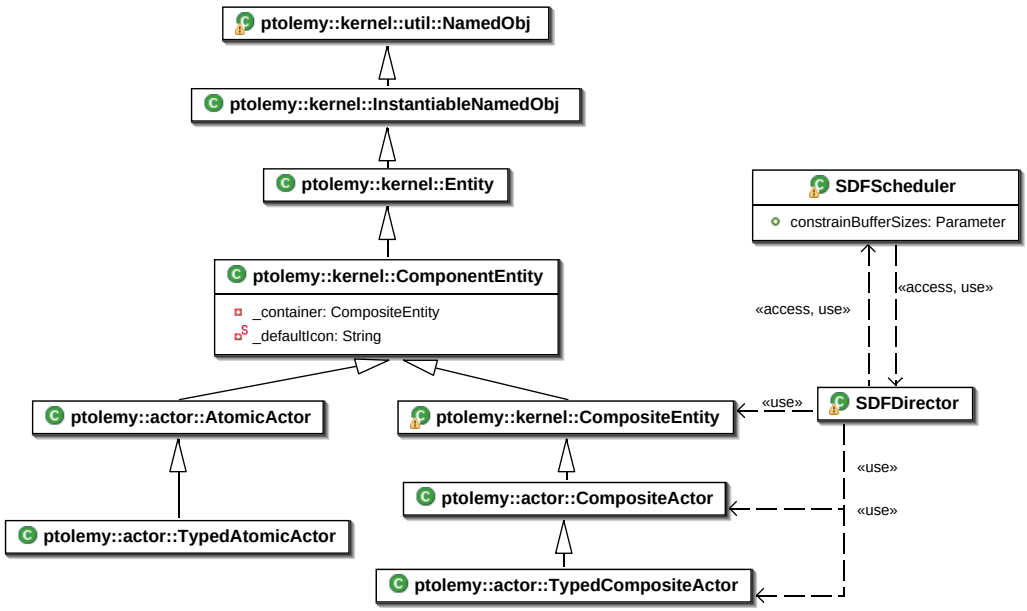


FIGURE 16: ENTITY CLASS HIERARCHY RELATED TO SDF DIRECTOR AND SCHEDULER.

Entities are aggregations of ports which they need to communicate with other entities. An entity can contain any number of `Port` instances. Derived classes may wish to constrain it to a subclass of `Port`. A `ComponentEntity` can contain instances of `ComponentPort`. Normally, a `Port` is contained by an `Entity`, although a `Port` may exist with no container. To represent a directed graph, entities can be created with two ports, one for incoming arcs and one for outgoing arcs. More

generally, the arcs to an entity may be divided into any number of subsets, with one port representing each subset. `ComponentPorts` support hierarchy. `IOPorts` support exchanging data between entities via message passing. It can serve as an input port, an output port or both. A `TypedOIPort` is an `IOPort` with a type. The `SDFIOPort` class extends `TypedIOPort` with convenience methods for handing the token production and consumption rates.

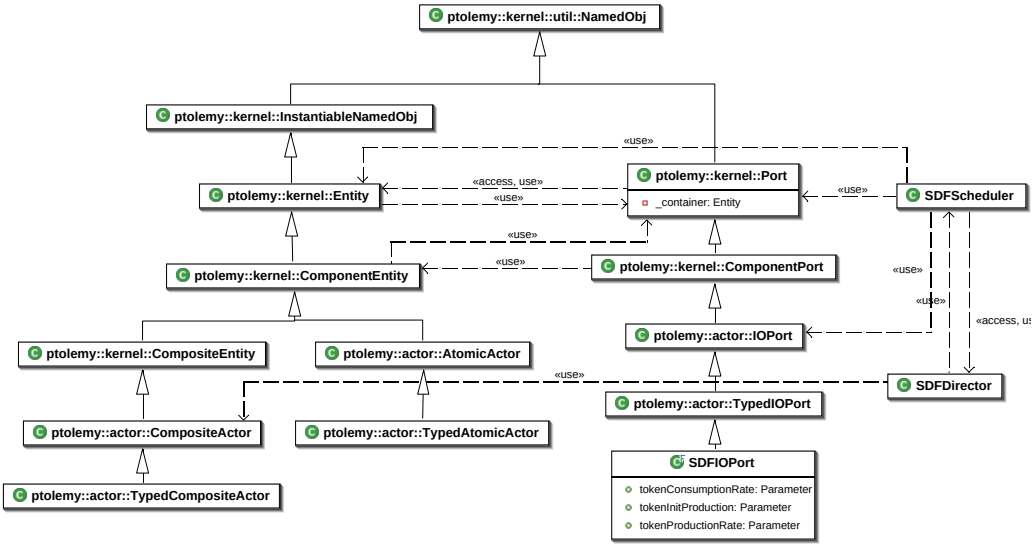


FIGURE 17: ENTITY AND PORT CLASS HIERARCHY.

Relations represent connections between ports and therefore relations between entities. A `Port` is the interface of an `Entity` to any number of `Relations`. The role of a `Port` is to aggregate a set of links to relations. A `Port` can link to any instance of `Relation`. A `ComponentPort` can have *inside* links as well as the usual *outside* links supported by the base class. That is, it can have links to relations on the interior. An inside link is a link to a relation that is contained by the container of the Port. A component port can link to any instance of `ComponentRelation`. A `ComponentRelation` defines a relation supporting hierarchy. An `IOPort` has a width, which by default is either zero or one. The width is the sum of the widths of the linked relations. A port with a with greater than one behaves as a bus interface, so if the width is w, then the port can simultaneously handle distinct input or output channels of data. An `IOPort` can only link to instances of `IORelation`. A `TypedIOPort` can only link to instances of `TypedIORelation`. Figure 18 shows the Port and Relation class hierarchy that from the infrastructure for the topology.

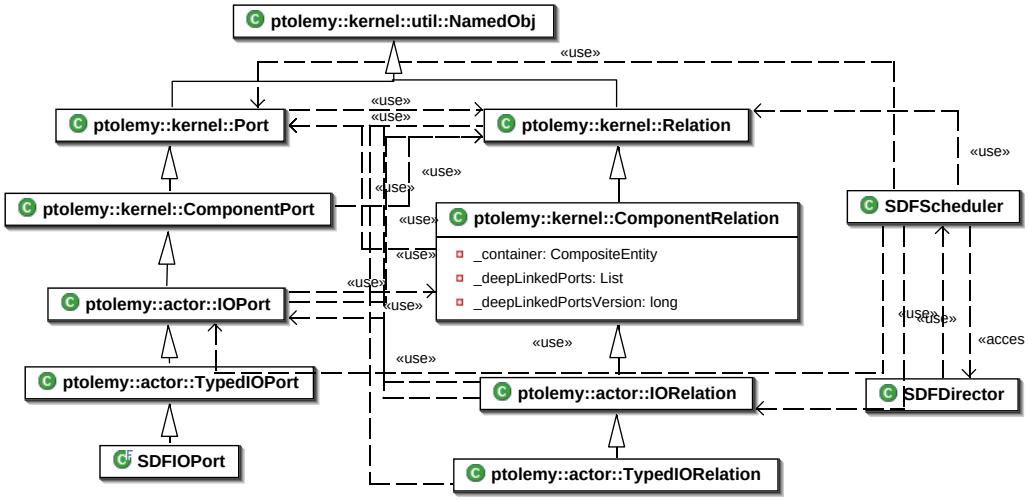


FIGURE 18: PORT AND RELATION CLASS HIERARCHY.

The `Receiver` interface defines methods for objects that can hold tokens. It declares two key methods: `put()` and `get()`. The `put()` method deposits a token into the receiver. The `get()` method retrieves a token from the receiver that has been put there before. The order of the retrieved tokens depends on the specific implementations, and does not necessarily match the order in which tokens have been put. Objects that implement this interface can only be contained by an instance of `IOPort`. `AbstractReceiver` is an abstract implementation of the `Receiver` interface. The domain-specific methods are left undefined. The `SDFReceiver` implements a first-in, first-out (FIFO) queue receiver with variable capacity and optional history. By default the capacity is unbounded, but it can be set to any nonnegative size.

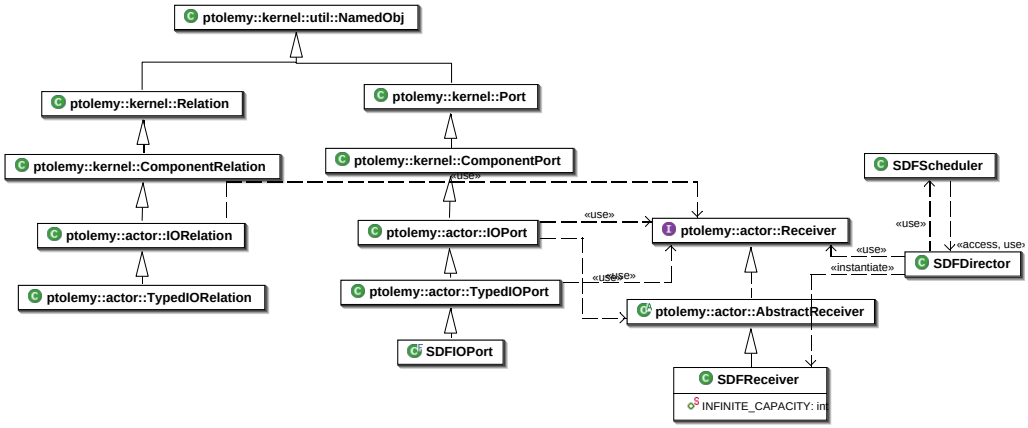


FIGURE 19: RECEIVER AND PORT CLASS HIERARCHY.

Figure 19 shows the Receiver and Port class hierarchy. If an `IOPort` is an input port, then it contains some number of receivers, which are responsible for receiving data from remote entities. If it is an output port, then it can send data to remote receivers. Its receivers are created by a director. It must therefore be contained by an actor that has a director.

5.3 EXTENSION OF THE SOFTWARE ARCHITECTURE

Once the underlying software architecture is explained, we describe the software architecture we have built on top of the existing one. For the extended architecture we document what we consider relevant features, for minor details we refer the reader to the documented source code, that can be found at the Ptolemy II official website [22]. In order to keep the existing architecture intact, we have created a new package `ptolemy.distributed` under which the source code that implements ADS can be found. Common features that can be reutilized to make distributed versions of other domains To enable further supporting implementations of ADS for other domains, new packages can be created under `ptolemy.distributed.domains` with a trailing *`.domainname`* for every given domain.

The package `ptolemy.distributed.domains.sdf.kernel` includes the core classes implementing ADS for SDF. The `DistributedSDFDirector` class extends `SDFDirector` and `DistributedSDFScheduler` extends `SDFScheduler`. A summary of the packages is included at the end of this chapter.

5.4 PARALLEL SCHEDULER

In static scheduled domains, computation of the schedule resides primarily in the scheduler that is an attribute of the director. Schedulers implement the scheduling policy that complies with the given MoC. The parallel scheduling approach has been described in Section 4. The class implementing the computation of PAPS is called `DistributedSDFScheduler` and can be found under `ptolemy.distributed.domains.sdf.kernel`. The implementation approach was developed using AsmL [104] and is described in [105][106]. The `DistributedSDFScheduler` extends the existing `SDFScheduler`. To allow extension, minor modifications had to be introduced in the existing `SDFScheduler`. The members: `_externalRates`, `_firingVector`, `_rateVariables` and methods: `_computeMaximumFirings`, `_countUnfulfilledInputs`, `_getFiringCount`, `_setFiringVector`, `_simulateExternalInputs`, `_simulateInputConsumption`, methods had their visibility changed from private to protected. A great deal of the existing code has been reused to avoid duplicates and four methods have been overridden to compute the topological sort and from it the parallel schedule.

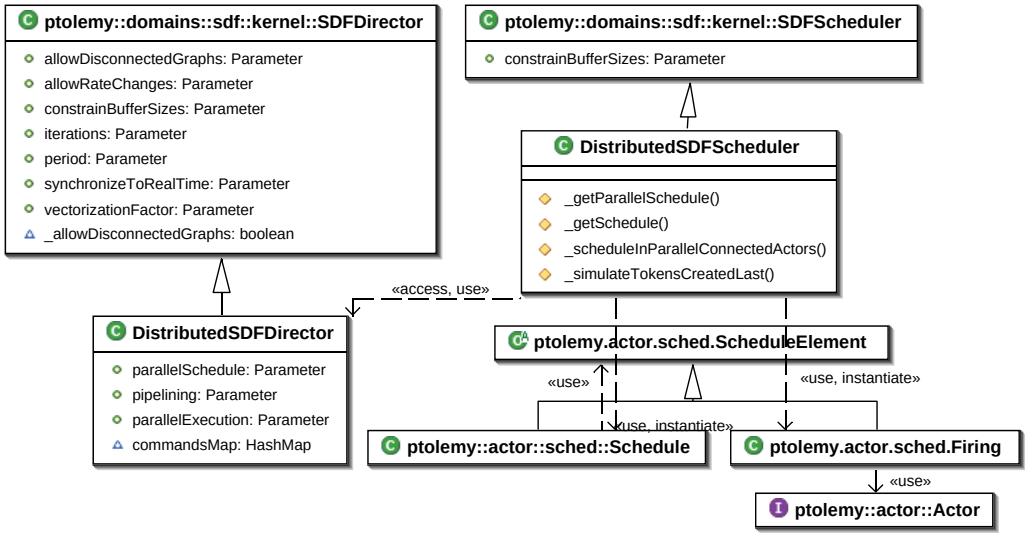


FIGURE 20: DISTRIBUTEDSDFSCHEDULER AND RELATED SCHEDULING CLASSES.

Figure 20 shows the `DistributedSDFScheduler` in relation with the `DistributedSDFDirector` and related classes involved in scheduling. The classes that implement schedules are shown as well.

To enable interaction with the user, the DistributedSDFDirector declares three parameters `parallelSchedule` `parallelExecution` and `pipelining`. They are shown in the configuration dialog for the Director allowing the user to decide whether a sequential or a parallel schedule should be computed, to enable distributed simulation and to enable pipelining as described in Section 4.11. If these parameters are set to false, the director behaves as the preexisting `SDFDirector`. If `parallelSchedule` is set to false, the `DistributedSDFScheduler` uses the parent's methods to calculate the schedule (in a sequential manner) otherwise calculates the parallel schedule. The `Schedule` class that represents a static schedule of an actor's executions supports parallel schedules. A `Schedule` is a list of `ScheduleElements`, which can be either a `Schedule` or a `Firing` of an `Actor`. A sequential schedule would typically contain a list of `Firings` of actors which are to be performed one after the other. For parallel schedules, we create a Schedule containing a list of `Schedules` each of them containing a list of `Firings`. It is a two-level structure that indicates that all the `Firings` contained in one of the `Schedules` inside the main schedule can be performed in parallel. That information is used by the director when dispatching the actor firings; it is explained in Section 5.9. Once the parallel schedule has been computed, we need the infrastructure to allow the parallel execution of the actors.

5.5 SERVER AND SERVICE

To constitute a distributed platform server software has to be running on a set of computers on a network. Such a server has to provide services for executing Ptolemy II actors and interfacing them with other actors and the director. This section describes the server and service software. Since we want the processing to be done on the server

side, our choice for the architecture is a thin Java RMI Proxy. The Java RMI API [107] performs the object equivalent to RPCs. Figure 21 shows the implementation model of Java-RMI. Common interfaces are shared by the client and the server. The stub on the client side is used to make calls to the implementation on the server.

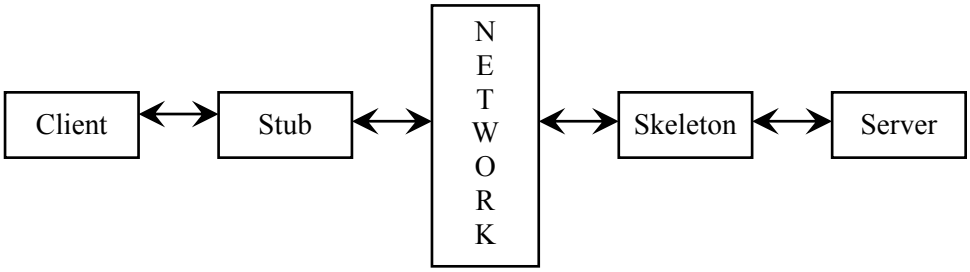


FIGURE 21: IMPLEMENTATION MODEL OF JAVA-RMI.

The proxy just exists on the client to take calls from the client, invoke the method in the service on the server, and return the result to the client.

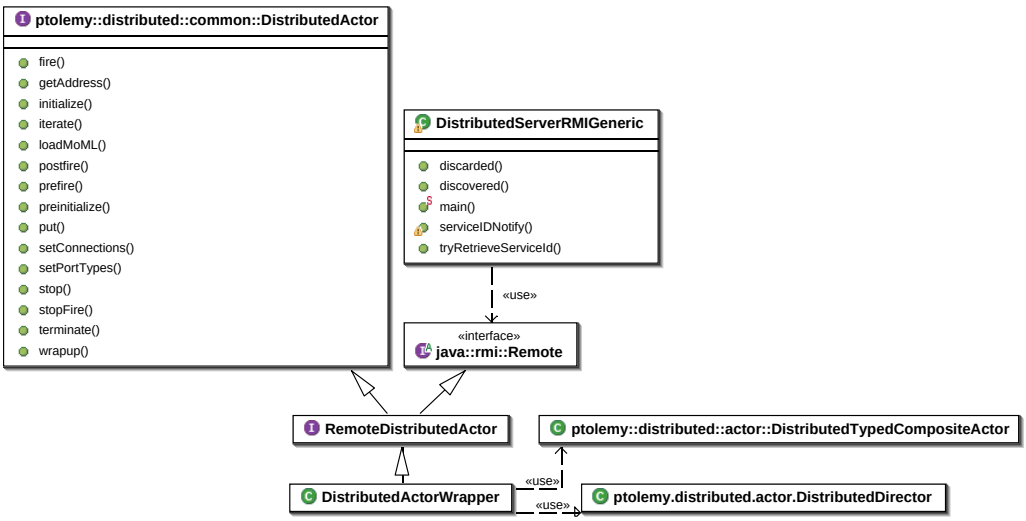


FIGURE 22: SERVER (DISTRIBUTEDSERVERRMIGENERIC) AND SERVICE (DISTRIBUTEDACTORWRAPPER) CLASSES.

Figure 22 shows the server (`DistributedServerRMIGeneric`) and service (`DistributedActorWrapper`) classes provided in the package `ptolemy.distributed.rmi`. The server class takes a configuration file name as an argument. The corresponding file specifies the service to be loaded and Jini and Java RMI configurations. Jini is a peer discovery protocol and is further described in the next section. After the service is loaded, the server registers it with the service locator and stays alive. The service that hosts Ptolemy II actors is `DistributedActorWrapper`. This class implements the `RemoteDistributedActor` interface which extends the `DistributedActor` and `Remote` interfaces. The `Remote` interface serves to identify the interfaces whose methods may be invoked from a non-local virtual machine. Only the methods specified in an interface that extends `Remote` are available remotely. `DistributedActorWrapper` has two main purposes: to expose the

`DistributedActor` interface to the outside to receive external RMI calls and to create an environment for the hosted actor that looks as if it was residing in the local machine. The `DistributedActor` interface includes all the action methods plus a few more to load actors from MoML [26] specifications and virtually connect to remote actors over the network. The `loadMoML()` method receives a string describing an actor in MoML. This string is processed to add extra MoML code that embeds it in a `DistributedTypedCompositeActor` with a DistributedDirector. The resulting MoML is parsed into a `CompositeActor`. For optimization, actors are instantiated from local storage, which presupposes that a copy of the class files exist on the server side. Figure 23 shows the architecture of the server.

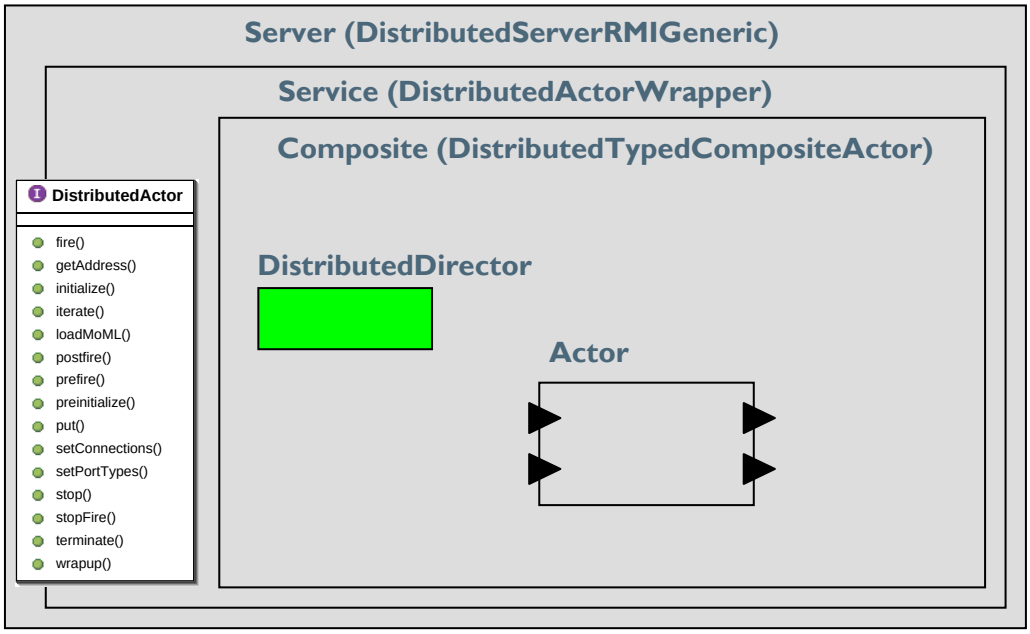


FIGURE 23: SERVER, SERVICE, COMPOSITE AND ACTOR.

For correct usage, the following steps should be followed in the specified order: loading the actor, `setPortTypes()` and `setConnections()`, action methods. This is enforced by the `DistributedSDFDirector`.

5.6 PEER DISCOVERY

Servers have to publish their services to make them available for clients. This is enabled by a peer discovery protocol. This section introduces our choice for discovery, Jini Network Technology [108]. Since Java is the primary programming language used in Ptolemy II, we have evaluated the two main java technologies that provide this feature JINI and JXTA [109]. Another open source alternative is Kademlia [110]. Since Jini is aimed at discovery and our distributed platform is a cluster of computers in a local network Jini is sufficient for our purposes. Jini has a less steep learning curve and outperforms Jxta [111].

Jini[2] is the name for a distributed computing environment that offers *network plug and play*. A device or a software service can be connected to a network and announce its presence, and clients that wish to use the service can then locate it and call it to perform tasks. Jini can be used for mobile computing tasks where a service may be connected to a network for a short time, but it can be used in any network where there is some degree of change. Jini supplies a middleware layer to link services and clients from a variety of sources. A Jini system or *federation* is a collection of clients and services all communicating by the Jini protocols. In our case, one client will locate several services to perform distributed computing and communicate via RMI calls. In a running Jini system, there are three main players as shown in Figure 24.

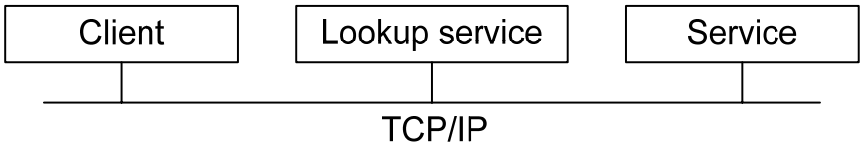


FIGURE 24: THREE DIFFERENT COMPONENTS IN THE JINI SYSTEM.

There is a *service*, in our case a `DistributedActorWrapper` as described above. There is a *client* which would like to make use of this service, in our case a `DistributedSDFDirector` wanting to perform distributed simulation. Thirdly, there is a *lookup service* (service locator) which acts as a locator between services and clients. The Lookup Service runs somewhere reachable for both client and services; it can even run on the same computers as client or servers. There is an additional component, and that is a *network* connecting all three of these, and this network will generally be running TCP/IP. (The Jini *specification* is fairly independent of network protocol, but the only current *implementation* is on TCP/IP).

5.6.1 THE LOOKUP SERVICE

The lookup service is a Jini service that is specialized in storing service and passing them on to clients looking for them. A client locates a service by querying a lookup service (service locator), but first services must register with it in order to be found. For clients and services to query and register they must first discover a lookup service. The lookup service (or set of services) is previously started by some independent mechanism and running on the network. There may be any number of lookup services running on the network accessible to broadcast search. On a small network, such as a home network, there may be just a single lookup service, but in a larger network there may be many. Each of these may choose to reply to a broadcast request.

5.6.2 DISCOVERY

The initial phase of both client and service is thus discovery. The search for a lookup service can be done either by unicast or multicast. Unicast discovery can be

[2] Jini is not an acronym for anything, and does not have a particular meaning (though it gained an interpretation of "Jini Is Not Initials").

used when the location of the lookup service is known in advance. This is expected to be used for a lookup service that is outside of your local network. The ADS implementation provides a way to specify locations of known service locators via the configuration file. If the location of a lookup service is unknown, it is necessary to make a broadcast search. Jini uses UDP support for multicast. Because multicast is expensive in terms of network requirements, most routers block multicast packets. This usually restricts broadcast to a local area network, although this depends on the network configuration and the time-to-live (TTL) of the multicast packets.

5.6.3 SERVICE REGISTRATION

Servers register services with lookup services to make them available. Figure 25 depicts server discovery and registration.

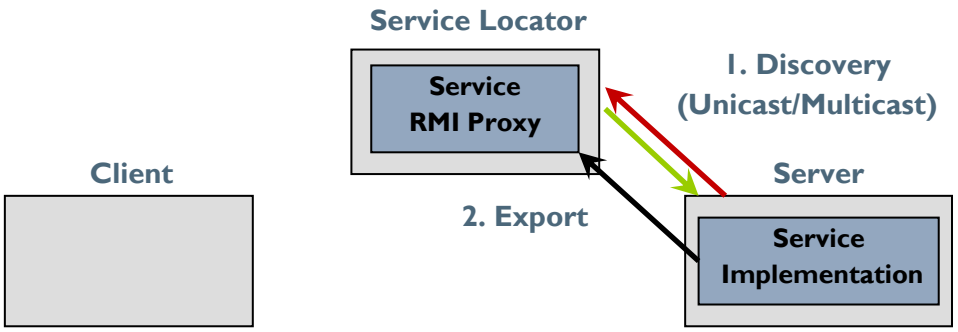


FIGURE 25: SERVER DISCOVERY AND REGISTRATION.

The following steps are performed once the server is started as a service provider. First, there is discovery to find a service locator. When a service locator is found, the server proceeds to registration. Registering the service can be done in different ways. We have chosen an RMI proxy since all the service work is done on the server side. For this, the server makes available a thin proxy that channels method calls from the client to the service implementation across the network and returns the result back to the client. Once the proxy is created, the server will register the service proxy with service locators and then wait for network requests to come in for the service. What the service provider exports as a service object is a proxy for the service. The proxy is an object that eventually runs in a client, and will usually make calls back across the network to service backend objects. These backend objects running within the server actually complete the implementation of the service. The proxy and the service implementation are tightly integrated; they must communicate using a protocol known to them both, and must exchange information in an agreed manner. The proxy copy is an *instance* of an object in serialized form. So what is stored on each service locator is an instance of a class. The service class is specified in the configuration file. Once the service (or better the proxy) is registered, the lease has to be maintained. Leasing is the mechanism used between applications to give access to resources over a period of time in an agreed manner. The class `JoinManager` supports all these features. It uses a `LookupDiscoveryManager` for the discovery and a

`LeaseRenewalManager` for the leasing. The proxy is previously created by an `Exporter` and the `JoinManager` handles the registration.

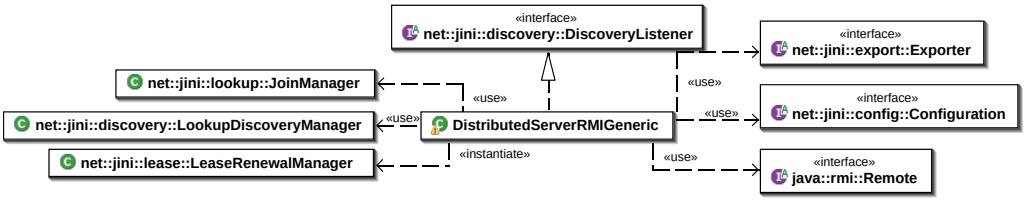


FIGURE 26: SERVER CLASS AND RELATED JINI CLASSES.

Finally the server is kept alive so that the service stays running.

5.6.4 SERVICE LOOKUP

The distributed platform to run simulations is a set of processes that execute in different machines. In principle the locations of such processes are not known by the client, meaning that they have to be discovered on the fly. In order to make this step transparent to the user, a peer discovery protocol is used. This way the `DistributedSDFDirector` can find the peers it needs for the execution of the actors without knowing them previously. Figure 27 shows the service lookup implementation architecture. The `DistributedSDFDirector` relies on the `ClientServerInterationManager` class from the `ptolemy.distributed.client` package to implement service lookup.

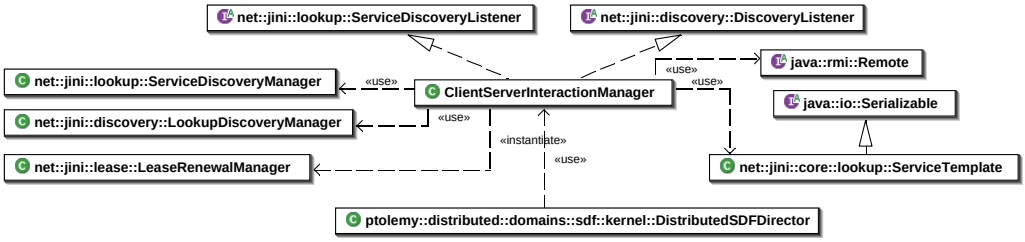


FIGURE 27: CLIENTSERVERINTERACTIONMANAGER CENTRALIZES CLIENT'S SERVICE LOOKUP.

The first step is to prepare for discovery by initializing Jini. The class `ClientServerInteractionManager` encapsulates Jini functionality. Here, the number of services required to run the simulation, equal to the number of actors in the model, is calculated. A configuration file is used to store information about various settings as for example unicast locators. Then, discovery of a lookup service is performed. A client (in this case the `DistributedSDFDirector`) locates a service by querying a lookup service (service locator). The `LookupDiscoveryManager` class makes it easier to find lookup services using unicast and multicast searches. Once a lookup service is found, we query to find services by their interface. The class `ServiceDiscoveryManager` is a utility class designed to help in service discovery. The `ServiceDiscoveryManager` can provide a cache of services since we want to make use of multiple instances of services (one for each actor). This cache looks after monitoring lookup services to keep the cache up-to-date with services. The `LookupCache` takes a template for

matching against interface. The cache maintains a set of references to services matching the template. In our case we are looking for services implementing the interface `DistributedActor`. The Lookup Service provides references to services matching the template provided. If servers fail, it might happen that their services remain registered with the service locator during the time between the crash and the lease expiring. To avoid using invalid services filter them to make sure that the references to services that will be used from this point on are alive. This is done by attempting to use every service and if succeeding; add it to a list of alive services.

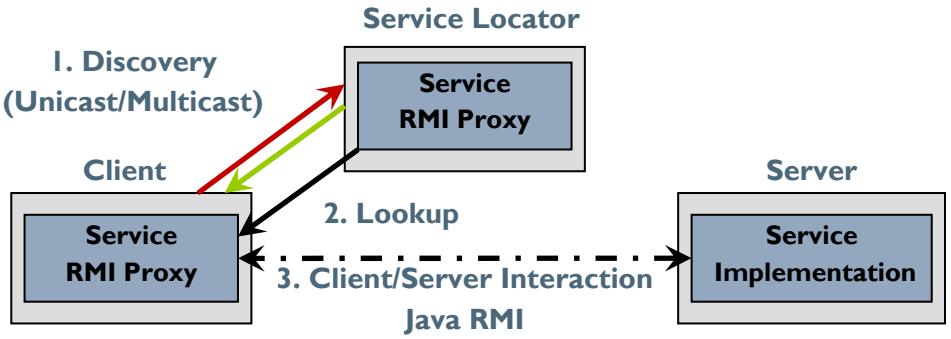


FIGURE 28: SERVICE LOOKUP AND CLIENT/SERVER INTERACTION.

Once we have gathered enough service proxies we are ready to interact remotely with the services. From this point we assume that the service stays alive, and we can interact with it. Several RMI calls to the different methods exposed by the service will be performed for deployment and to virtually set up the topology. The calls have to be performed in a particular order to be correct, which is enforced by the `DistributedSDFDirector`.

5.7 DEPLOYMENT

With a sufficient number of service proxies, deployment of actors can take place. Client threads are created for every actor that is deployed. Each contains a reference to a service. `ClientThreads` allow for real parallel execution and are further explained in Section 5.9. Actors are mapped on the threads that control access to the services. The mapping is trivial in the sense that no effort is made to optimize the mapping. Once mapped, actors are deployed on the services according to the mapping. A MoML description of the current status of the actor is sent via an RMI call to loadMoML() to the associated service. The conversion of the actor on to MoML is performed after the model has been initialized thereby transferring the current state of the actor to the remote side. For efficiency, the actor is instantiated from a local copy of the implementing class.

5.8 DISTRIBUTED MESSAGE PASSING

After all the actors are deployed, the topology is deployed. The existing message passing mechanism relies on execution on a local machine and in shown in Figure 29.

When an actor performs a send on an output port, receivers at the remote side are gathered and a copy of the data token is put into each one of them. Receivers are created at every connected input port to hold data tokens for every connection implementing a FIFO queue. Every Receiver can be identified in the local machine via a memory reference (pointer) but when moving to a distributed environment it is not possible.

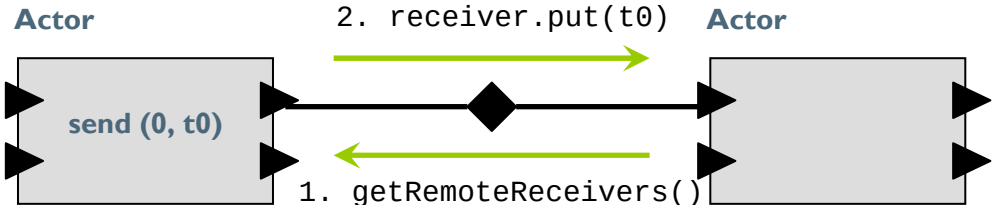


FIGURE 29: MESSAGE PASSING IN PTOLEMY II.

To solve this problem, we have created two new types of receivers. `DistributedSDFReceiver`, extends `SDFReceiver` with an unique ID to identify receivers in the distributed platform. DistributedSDFDirector creates receivers of this type. `DistributedReceiver` is a more generic receiver that forwards tokens produced in output ports to remote services whenever its `put()` method is called. Since ports are connected to relations we have created a special type of relation called `DistributedTypedIORelation` It overrides the `deepReceivers()` method that returns the connected receivers to this relation. In this case, the relation only contains (is connected to) only one `DistributedReceiver` in charge of forwarding tokens to the distributed services that are connected.

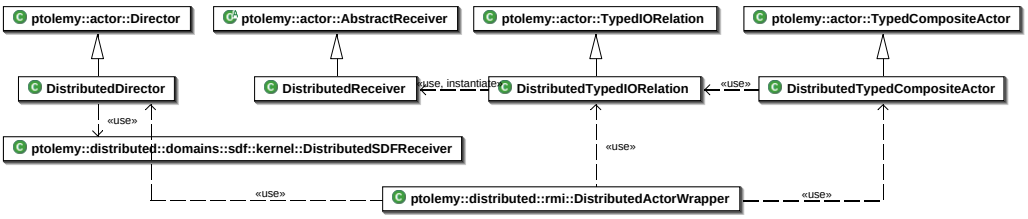


FIGURE 30: SERVICE INTERNAL CLASSES.

The deployment of the topology is performed by interconnecting all the remote actors in the same manner as the original model's topology. In other words, the connections defined by the model's topology are created virtually over the distributed platform.

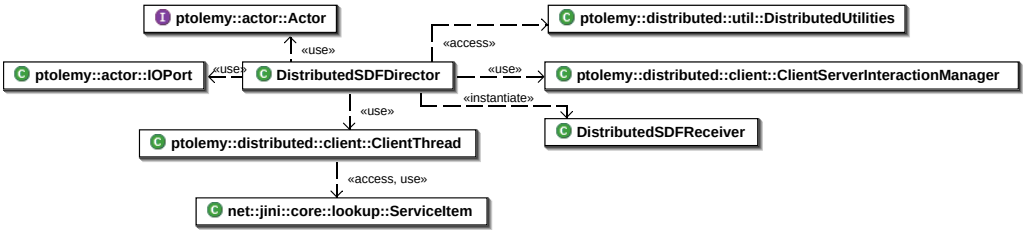


FIGURE 31: DEPLOYMENT CLASSES ON THE CLIENT SIDE.

For each actor, a mapping of receivers to ports is created. It represents for a given port the receivers it contains. In case the port is an input port it consists of a set of receivers ID's i.e. (inputport, (ID_1, ..., ID_n). In case of an outputport, it contains a map of services to receiver's IDs, i.e. (outputport, (($service_1$, (ID_1, ..., ID_i), ..., ($service_n$, (ID_j, ..., ID_r))). The services for the output ports are copies of the RMI proxy thus allowing access to the remote service when instantiated. This information is gathered from the topology and sent over to the corresponding service for all its ports. The types of the port are also set on the remote actor.

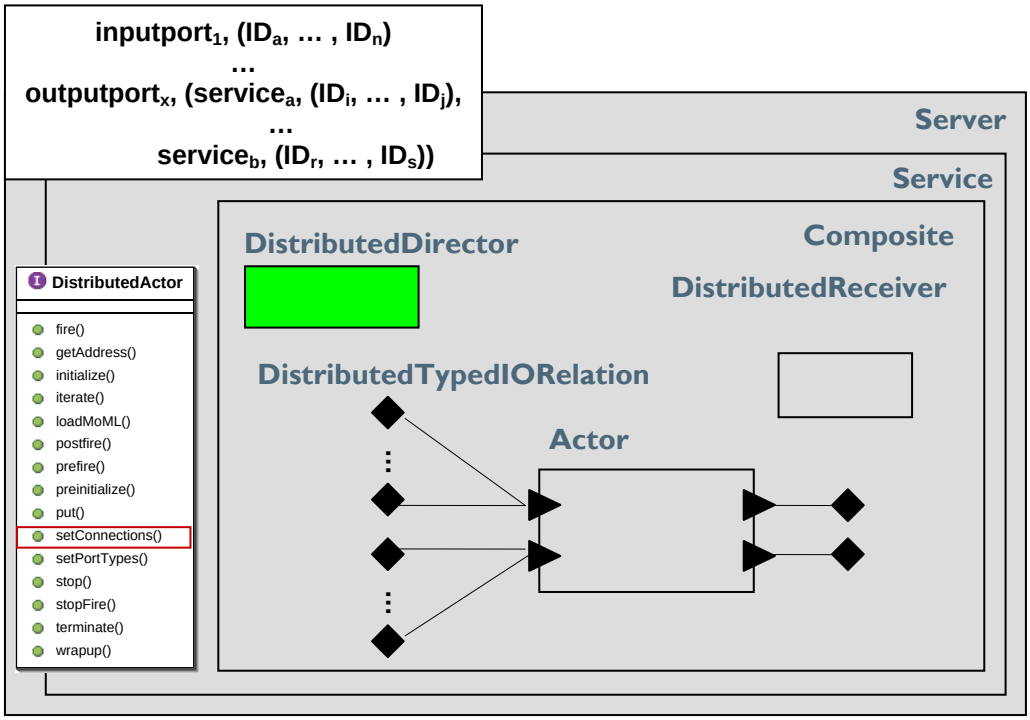


FIGURE 32: SERVICE INTERNAL ARCHITECTURE.

When the `setConnetions()` call is received at the service side it must contain information concerning virtual connections concerning the wrapped actor. This is guaranteed by the `DistributedSDFDirector`. The connections mapping contains a list of ports of the actor, and for each of them a mapping depending on the type of port. For the input ports, a new relation of the type `DistributedTypeIORelation` is created for every ID that is received. This is to force a `Receiver` to be created whenever `createReceivers()` in the corresponding port is called. The list of IDs is passed to the `DistributedDirector` to create the receivers with the correct values. For the output ports, only one relation of the type `DistributedTypedIORelation` is created and the mapping of services to IDs is set into the relation. The relation sets that information in its contained `DistributedReceiver` that is in charge of token forwarding whenever send is called on the port (and then put on the receiver). A copy of the token sent to every service with the list of IDs meant to receive it. This is depicted in Figure 32.

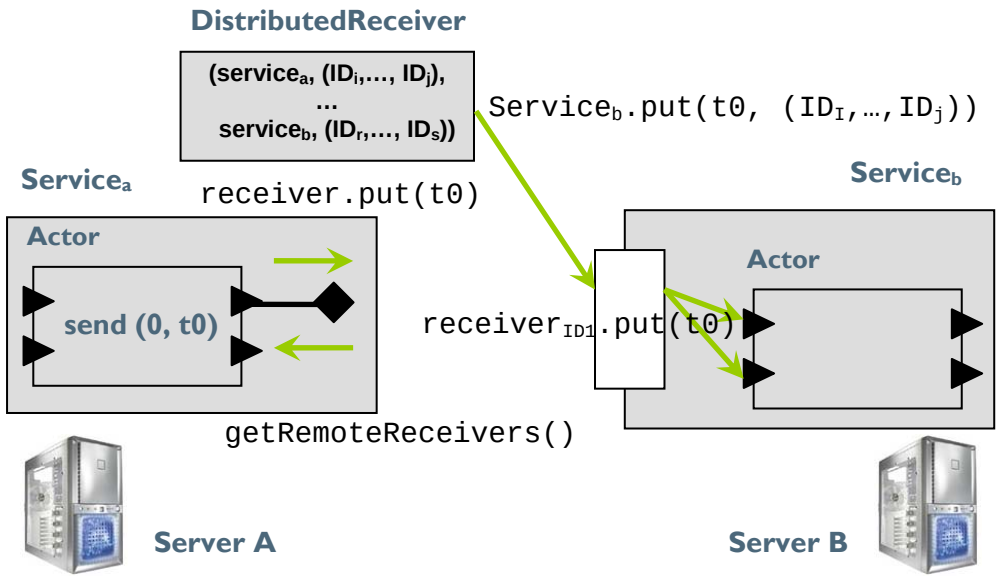


FIGURE 33: DISTRIBUTED MESSAGE PASSING (DECENTRALIZED).

Figure 33 shows the distributed message passing mechanism. When an actor running in a distributed service sends a token over a port, the token is received by a `DistributedReceiver`. The receiver contains a mapping of services and IDs that are meant to receive a copy of the token. The receiver then sends a copy to every service on the map with an attached list of IDs that receive the token. On the remote side, a token and a list of IDs are received. A copy of the token is delivered to each receiver in the ID list. This is a decentralized asynchronous message passing implementation.

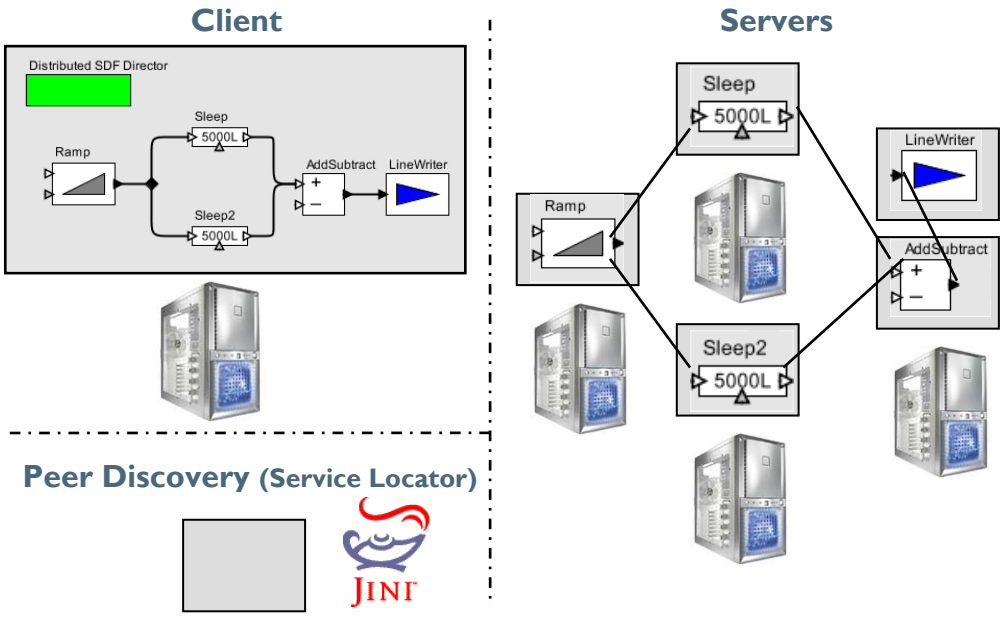


FIGURE 34: DEPLOYMENT.

5.9 PARALLEL DISPATCHING AND SYNCHRONIZATION

Once the model (actors and topology) is deployed to the distributed platform, execution is started. In order to issue commands in parallel a parallel dispatching and synchronization mechanisms is required. This section describes our implementation. At the deployment of actors, a collection of threads are created to control remote services each of them containing a RMI proxy to a service.

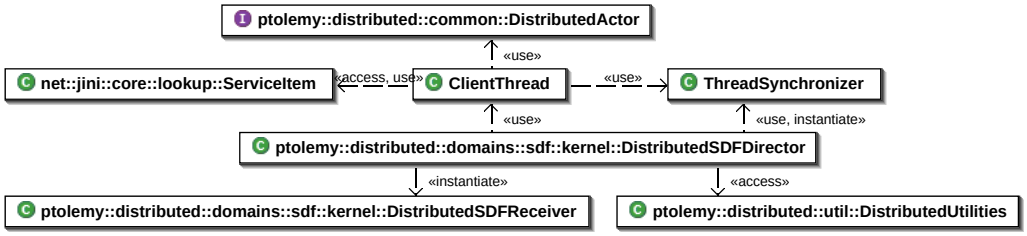


FIGURE 35: PARALLEL DISPATCHING AND SYNCHRONIZATION CLASSES.

`ClientThreads` allow simultaneously (or pseudo-simultaneously) running taks therefore enabling parallel issuing of commands to remote services. `Clienthreads` prevent the client's main thread of execution to be blocked by the remote calls to the remote services. `ThreadSynchronizer` synchronizes access to the commandsMap, a mapping used to issue a set of commands at once. In order to allow parallel execution of commands, the `ClientThreads` that manage remote actors locally in the `DistributedSDFDirector` have to access the commands without blocking the main thread in a synchronized manner. Commands are represented by integers. It provides mechanisms to issue sets of commands and synchronize the access to those commands by the client threads. It is assumed that no new set of commands is issued before the previous set of commands has been processed. Every ClientThread is responsible to set itself as ready after performing a command.

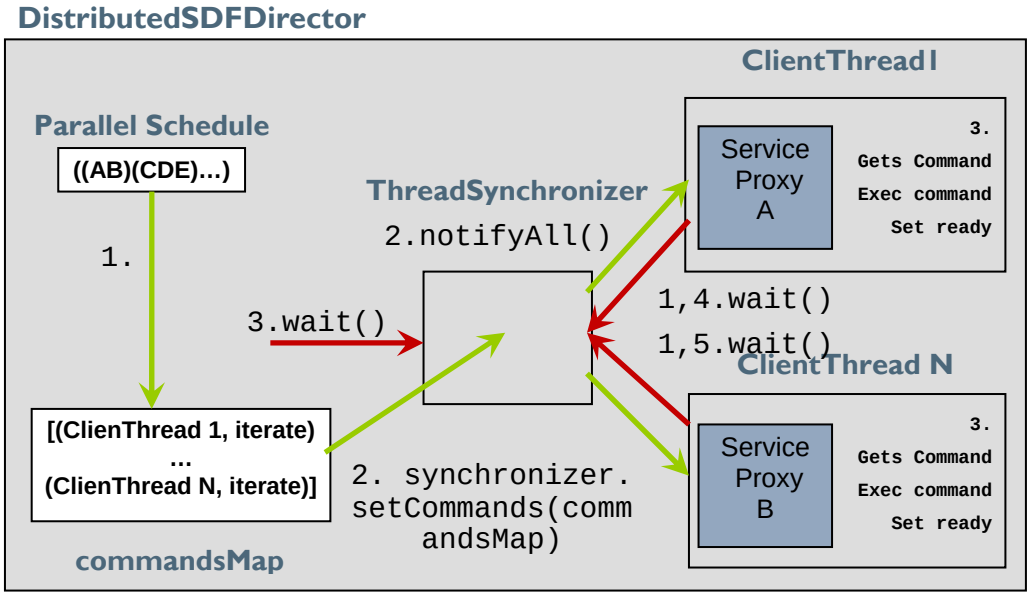


FIGURE 36: PARALLEL DISPATCHING AND SYNCHRONIZATION MECHANISM.

Figure 36 shows the dispatching and synchronization mechanism. It takes the parallel schedule one level after the other so every subschedule corresponds to a level in the DAG and a step in the execution. In the initial state, all the `ClientThreads` are running, blocked on the `ThreadSynchronizer` waiting for the commands to be issued. For each level, the client first creates a map of commands that can be issued in parallel. The commands are set into the ThreadSynchronizer that notifies all the blocked threads. The threads awaken and if there is a command for them they execute it and set ready for another command. Meanwhile, the `ThreadSynchronizer` blocks waiting for all the threads to finish and notify that they are ready for another round. This mechanism sets an execution barrier that ensures that no set of commands is issued before the previous is consumed. In other words, the parallel schedule is executed by levels. This ensures correctness of the simulation.

If the simulation uses pipelining, a buffering phase is introduce after initialization that fills the queues with data tokens so that a fully parallel execution can be performed. For a given level of the schedule, the buffering phase fires all the actors on that level and the previous ones. When the last level of the schedule is reached a fully parallel execution is achieved.

Parallel Schedule: ((A)(BC)(D)) Iterations: 5			
Step	No Pipelining	Pipelining	
1	A	A	Buffering
2	BC	ABC	
3	D	ABCD	
4	A	ABCD	
5	BC	ABCD	
6	D	BCD	Completion
7	A	D	
8	BC		
9	D		
10	A		
11	BC		
12	D		
13	A		
14	BC		
15	D		

FIGURE 37: STEPS FOR A GIVEN PARALLEL SCHEDULE WITH AND WITHOUT PIPELINING.

Figure 37 shows the steps taken when dispatching the same schedule with and without pipelining. The comparison shows a big difference in the number of steps taken directly related with performance.

5.10 SOFTWARE PACKAGES

The ADS software implementation is organized under `ptolemy.distributed` in the following packages:

- **`ptolemy.distributed.actor:`** Contains the following classes:

`DistributedDirector:` A Director governs the execution of a CompositeActor in a Distributed environment.

`DistributedReceiver:` An implementation of the Receiver interface for distributed environments.

`DistributedTypedCompositeActor:` An extension of `TypedCompositeActor` for distributed environments.

`DistributedTypedIORelation`: Extension of `TypedIORelation` for distributed environments.

- **`ptolemy.distributed.actor.lib:`** Contains actors specific for distributed enviroments.

  `DistributedLineWriter:` An actor that writes the value of string tokens to the standard output, one per line.

- **`ptolemy.distributed.client:`** Contains the following classes:

  `ClientServerInteractionManager`: Manager that handles and eases the discovery of services using JINI.

  `ClientThread`: Thread created at the client side to allow issuing of parallel commands to the servers.

  `ThreadSynchronizer`: A synchronizer for the client threads.

- **`ptolemy.distributed.common:`** Contains the following classes:

  `DistributedActor`: Interface for distributed actors.

- **`ptolemy.distributed.config:`** Contains configuration files for client and servers.

- **`ptolemy.distributed.demo.Sleep:`** Contains the Sleep demo and scripts to start the platform.

- **`ptolemy.distributed.domain.sdf.kernel:`** Contains the core classes implementing `DistributedSDF` with ADS. It contains the following classes:

  `DistributedSDFDirector`: Director for the distributed version of the synchronous dataflow model of computation implementing ADS.

  `DistributedSDFReceiver`: Extends `SDFReceiver` with an ID.

  `DistributedSDFScheduler`: A Scheduler for the DistributedSDF domain that computes topological sorts and computes a parallel schedule.

- **`ptolemy.distributed.jini:`** Contains Jini related files and scripts to start jinni services.

- **`ptolemy.distributed.jini.config:`** Contains the following classes:

- **`ptolemy.distributed.jini.jar:`** Contains the Jini class files contained in jar files required for running Jini. These are:

  `jini-core.jar:` Contains the major packages of Jini.

  `jini-ext.jar`: Set of packages which are not in the core, but are heavily used.

  `jsk-platform.jar, jsk-policy.jar`: Security related.

  `reggie.jar, reggie.dl.jar`: Lookup service implementation, server and client side.

  `sun-util.jar`: Convenience classes that are not essential but useful.

  `start.jar, tools.jar`: Different tools.

- **`ptolemy.distributed.rmi:`** Contains the following classes:

  `DistributedActorWrapper`: Wrapper for a distributed actor.

  `DistributedServerRMIGeneric`: A distributed server to execute ptolemy actors in a distributed manner.

  `RemoteDistributedActor`: A distributed executable entity that is accessed via RMI.

- **`ptolemy.distributed.util:`** Contains the following classes:

  `DistributedUtilities`: Utilities for the distributed package.

5.11 SUMMARY

We have presented a framework that enables ADS of SDF Ptolemy models. It features the following:

Correctness. Correctness means the precedence constraints in the model are preserved. Correctness is guaranteed by two mechanisms: A parallel schedule that generates PAPS as explained in section 4.10 and a dispatching mechanism that synchronizes the different execution steps of the given PAPS.

Optimality. ADS for SDF exploits all available parallelism in the models.

Memory limitations minimized. Bigger models are allowed by efficiently distributing memory load to PEs. Not only computers but also JVMs have memory limitations. Since different actors are deployed to different machines, memory consumption is shared among the PEs. Limitations become smaller the more we scale the platform. Memory consumption due to data traffic is also minimal unless vectorization of data is used.

Automation. Distribution is automated in a manner transparent to the user, avoiding the tedious and error prone task of parallelizing models. ADS relies on the actor model to provide partitioning and provides scheduling and dispatching mechanisms that optimally exploit the parallelism of the model. The deployment of

the simulation and the communication and coordination mechanisms are fully automated.

The different elements and their roles (client, server, service locator) have been introduced. The distributed platform on which to run the simulations and the software that creates such platform has been described and implemented in harmony with the existing architecture. Peer discovery enables finding the required resources to run the simulation. Deployment of actors is optimized by sending MoML descriptions of the actors and loading the classes from local storage. A distributed message passing mechanism is implemented that overcomes identification of receivers at remote locations. The routing of data is decentralized for optimization. Finally a parallel dispatching and synchronization mechanism enables exploiting parallelism in the models in a correct manner. Pipelining techniques optimize simulation and enabling sequential models to benefit from ADS.

Simulation speedup by exploiting parallelism and iterative nature of SDF models is explored in Chapter 6 where we evaluate performance of the implementation via experiments.

CHAPTER 6

EXPERIMENTS AND RESULTS

If your experiment needs statistics,
you ought to have done a better
experiment.
Ernest Rutherford

Experiments give empirical evidence of a thesis. In this chapter we describe and perform experiments to measure the efficiency of simulations using ADS versus the original sequential implementation. In order to substantiate the value of parallelization, we carefully make some assumptions, choosing a set of parameters of interest and designing and building models and test cases that will reflect properties relevant for this work. First we describe the building blocks for the models and the different parameters to study, as for example the topology of the model. We use artificial models that suit the parameters we want to explore. We acknowledge the value of models of real applications or systems. Unfortunately, even though there are known industrial cases, for reasons of confidentiality they cannot be made available for performance tests.

Next we describe the different test cases, and present the results. Speedup can be as a result of executing actors in parallel (those at the same level in result of the topological sort). But also exploitation of the number of *iterations* and *vectorization factor* may increase the efficiency of the model. The different test cases explore these sources of parallelism.

Moreover we provide expressions to compute the theoretical execution time. These expressions are used to calculate the relative overhead introduced by the implementation. That is the percentage of the overall makespan introduced by ADS. The extra overhead associated with using multiple processors can eat into the potential speedup of parallelized code. Experiments show small percentages of overhead proving efficiency of the implementation.

Finally we test the effect of granularity in performance and come up with some conclusions.

6.1 BUILDING BLOCKS FOR THE MODELS

Among the large variety of available actors supplied with Ptolemy II we have chosen the `Sleep` actor as our basic building block for the models used for the experiments. The `Sleep` actor delays the input token for a certain amount of real time. This is very convenient to allow comparison of performance, as it provides a way to represent the amount of computation that an actor is performing. Having actors with a settable variable complexity would be an option but heterogeneity of processors or different work loads among the processors would produce different results. Controlling the amount of real time the execution takes is a more homogeneous and reliable solution, since we prefer completion time over functionality. However, informal functionality tests were performed with simple models throughout the development and testing phases; alas they are not documented. Moreover, all the simulations performed in the experiments we report in this chapter have been checked for correctness. Correctness of the simulations is enforced by the computation of a correct schedule and its correct dispatching.

The `Sleep` actor is included in the package `ptolemy.actor.lib`. In Vergil's library of components, it can be found under *real time* in the standard actor library. The `sleepTime` parameter specifies the amount of real time (in milliseconds) the inputs are delayed.

One of the preconditions to benefit from a distributed simulation is that the actors are computationally expensive, such that the communication overhead becomes insignificant. Therefore, the value used for the `Sleep` actors in the following experiments with graph topology, number of blocks and number of iterations is 5 seconds. A later Section 6.4 investigates the effect of varying the sleep time for selected values of the main parameters.

6.2 PARAMETERS

A major factor in gaining from parallel execution is the *topology* of the model; it should allow parallel execution of the actors. Graphs, however, come in many shapes and we cannot consider all of them, because the number of experiments then grows exponentially in the number of actors. Therefore we have chosen two extreme cases and an intermediate one: The sequential (*seq*) topology contains a number of Sleep actors that are linked one after the other in a sequential manner. In the parallel (*par*) topology, all the Sleep actors can be executed in parallel, and in between there is a binary tree topology (*tree*), where every Sleep actor is connected to two Sleep actors in its output port.

The two other factors we want to investigate are less complex: The *number of blocks* (actors), and the *number of iterations* of the model. Based on some initial test, we have selected values for these that give us a fair indication of the interesting points, where parallelization starts to pay off.

These experiments thus explore three different parameters, while the fourth, *block size*, is kept constant. Table 1 shows a list of them, with a description and the set of values used.

TABLE 1: PARAMETERS EXPLORED IN THE EXPERIMENTS.

Parameter	Definition	Values
n	Number of iterations	1,5,10,20
m	Number of blocks	1,5,10,15 (tree 2,6,14)
g	Topology	seq, par, tree
b	Block Size (s)	5

For the binary tree topology we have chosen different number of blocks that correspond to the different number of blocks required to fill completely a parallel level.

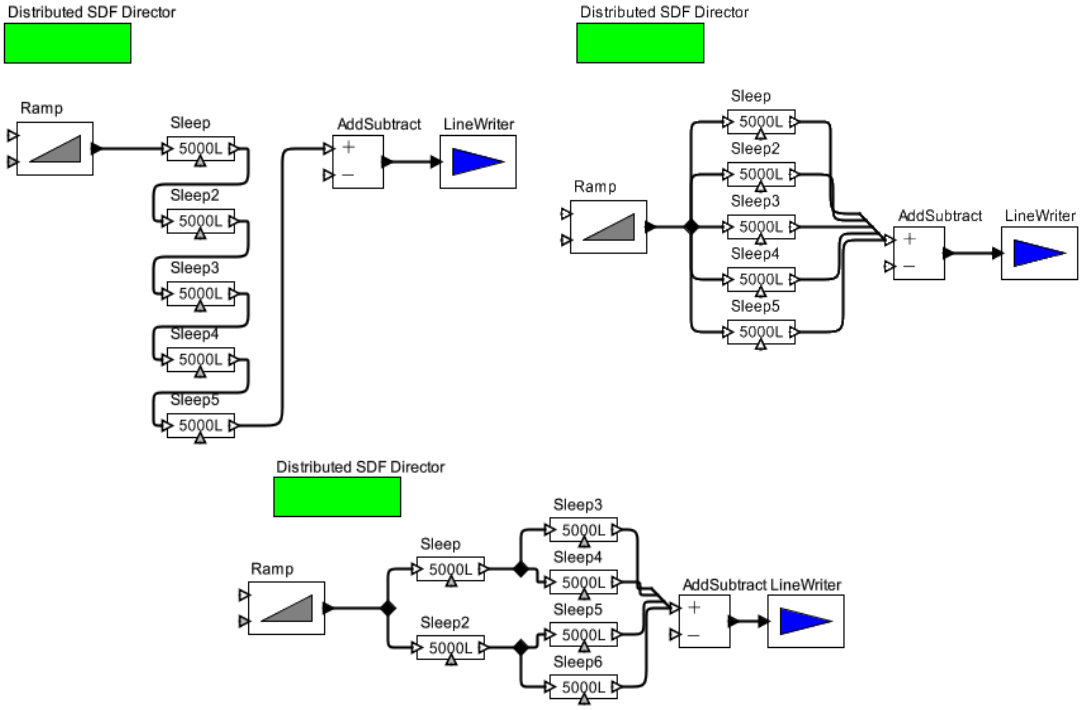


FIGURE 38: SEQUENTIAL, PARALLEL AND TREE TOPOLOGIES.

The `LineWriter` actors write the results to a file. Collecting the results from the leaves can be done by knowing the location of the file since the cluster runs AFS [112] (Andrew filesystem). AFS is a distributed networked file system which primary use is distributed computing. One of its major objectives is to make the way users retrieve information the same from any location.

6.3 TEST CASES

Three different sets of experiments have been performed. For each of them we calculate beforehand an analytical expression (a hypothesis) for the expected results. Those hypotheses are then checked against the empirical results. The three sets of experiments are:

- Sequential execution (*s*).
- Parallel execution without pipelining (*p*).
- Parallel Execution with Pipelining (*pp*).

Every test case has been repeated 9 times and the values in seconds have been rounded up. The results we present are averages of them. The variance of the values lies within a few seconds and thus much lower than the values, so we have not considered this further.

The hypotheses and the results are described in the following.

6.3.1 SEQUENTIAL EXECUTION

Sequential execution is the standard provided by Ptolemy II. A sequential schedule that preserves dependencies is executed on a local machine. For all three topologies described before, the expected execution time T is shown in equation (6-1). The equation is a result of fact that every block has to be executed by the single processor in sequence:

$$T_g^s[n,m] = n * m * b + \delta_g^s[n,m] \qquad \textbf{(6-1)}$$

The resulting expression of the execution time is dominated by *n, m* and *b*. $\delta^s_g[n,m]$ represents various overheads that include among others, initialization, scheduling and communication. In Table 2, we show the results for the experiments with sequential execution for all the topologies and different number for actors and iterations as shown in Table 1.

TABLE 2: PERFORMANCE RESULTS FOR THE SEQUENTIAL EXECUTION.

Model	Makespan per nr. of iterations (s)			
	1	**5**	**10**	**20**
M1S/P	5	25	50	100
M5S/P	25	125	250	501
M10S/P	50	250	501	1002
M15S/P	75	376	752	1504
M2T	10	50	100	200
M6T	30	150	301	601
M14T	70	351	701	1401

From the table we observe that as expected, execution times increase with the number of actors as well as with the number of iterations. In order to obtain the relative overhead, we compute the following:

$$\rho_g^s[n,m] = \frac{T_g^s[n,m] - (n * m * b)}{T_g^s[n,m]} \qquad \textbf{(6-2)}$$

The resulting values are plotted in Figure 39. It can be observed that they are in all cases insignificant. The tendency of the relative overheads is to decrease with the

number of actors and number of iterations. And "island" of higher values can be observed due to the effects of rounding the values and averages. However, for all practical purposes the overhead is insignificant, less than *0.5%* for a large block size.

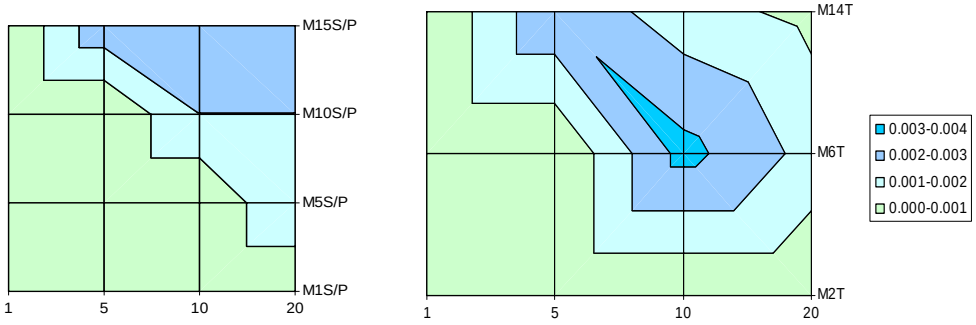


FIGURE 39: RELATIVE OVERHEAD FOR SEQUENTIAL EXECUTION.

Once we have obtained the data for the existing Ptolemy implementation of the SDF domain, we proceed to perform the experiments with ADS on the distributed platform. The results obtained so far will be used as a reference for comparison with the results obtained with the new implementation.

6.3.2 PARALLEL EXECUTION WITHOUT PIPELINING

Parallel execution without pipelining executes the parallel schedule one level after the other without pipelining. The expected execution times are as follows:

$$T^{p}_{seq}[n,m] = n*m*b + \delta^{p}_{seq}[n,m] \quad \textbf{(6-3)}$$

$$T^{p}_{par}[n,m] = n*b + \delta^{p}_{par}[n,m] \quad \textbf{(6-4)}$$

$$T^{p}_{tree}[n,m] = n*(\lceil \log_2(m+2) \rceil - 1)*b + \delta^{p}_{tree}[n,m] \quad \textbf{(6-5)}$$

$\delta^{p}_{g}[n,m]$ represents various overheads that include among others, initialization, scheduling and communication in the distributed without pipelining case. One of the main changes in this overhead is that communication is performed over the network. We expect this to increase the overhead, and we are especially interested in the extent to which this happens. For the sequential topology we define the same expression as for the sequential executions. For the parallel topology, the number of actors does not influence the results therefore *m* disappears from the expression. For the tree model, iterations last a number of times the block size, determined by the levels of the schedule. Due to the topology, this turns out to be $\lceil \log_2(m+2) \rceil - 1$. In Table 3, the results for the 3 different topologies are shown.

TABLE 3: PERFORMANCE RESULTS FOR THE PARALLEL EXECUTION WITHOUT PIPELINING.

Model	Makespan per nr. of iterations (s)			
	1	5	10	20
M1S	7	28	53	105
M5S	27	129	25	511

M10S	53	257	511	1015
M15S	79	384	764	1520
M1P	7	27	52	103
M5P	7	28	53	104
M10P	8	28	54	106
M15P	9	29	56	107
M2T	7	27	53	104
M6T	12	53	104	206
M14T	18	80	156	308

The parallel and tree models achieve a very significant reduction in execution time. As expected, the parallel model gets the most benefit out of the parallel execution as described in the equations (6-3) to (6-5). Comparing these results with the sequential execution, we can observe that for the sequential model, the results are even worse due to the communication overhead. This was as well expected. Again, we process the data to calculate the relative overhead:

$$\rho^{p}_{seq}[n,m] = \frac{T^{p}_{seq}[n,m] - (n*m*b)}{T^{p}_{seq}[n,m]} \qquad \textbf{s(6-6)}$$

$$\rho^{p}_{par}[n,m] = \frac{T^{p}_{par}[n,m] - (n*b)}{T^{p}_{par}[n,m]} \qquad \textbf{(6-7)}$$

$$\rho^{p}_{tree}[n,m] = \frac{T^{p}_{tree}[n,m] - (n*(\lceil \log_2(m+2) \rceil - 1)*b)}{T^{p}_{tree}[n,m]} \qquad \textbf{(6-8)}$$

The overheads for the various topologies are shown in Figure 40, Figure 41 and Figure 42. All of them show low values (<10%) for high values of *n*. Figure 40 shows the relative overheads for the sequential model. In this case, the values are rather small in general. They very quickly (with the increase of *m* and *n*), drop down to values that are irrelevant for the overall makespan. The sequential model presents a communication load that is more spread over time as compared with the models with higher degree of parallelism. Only one actor is fired at a time in this case. This results in low traffic both among the client and the servers and the servers themselves.

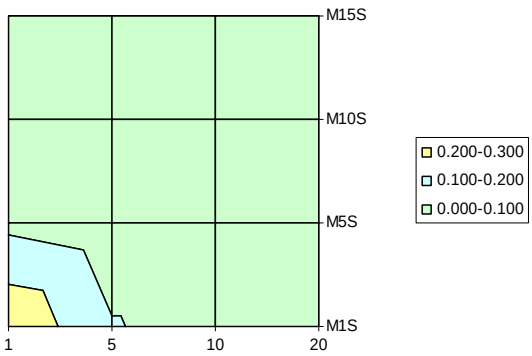


FIGURE 40: RELATIVE OVERHEAD FOR THE SEQUENTIAL MODEL AND PARALLEL EXECUTION WITHOUT PIPELINING.

The relative overheads for the parallel model are shown in Figure 41. The results are higher as compared with the sequential topology. Notice that the makespan values in this case are significantly smaller than in the sequential case, resulting in greater relative overheads. Moreover, a high number of actors results in a higher relative overhead as compared with the sequential topology. This can be explained by different reasons:

- **Higher Traffic:** Even though the same amount of traffic is generated in the overall execution of both the sequential and parallel models, in the parallel case, that traffic happens in a much shorter timespan as can be seen from the results.
- **Client CPU and Network Card Bottleneck:** The synchronization messages between the client and the servers happen all at the same time. The `ClientThreads` that allow for parallel dispatching compete for the same CPU and the same network card device to send requests to the servers when a set of commands is issued. The CPU and the network card are sequential devices. The more there are, the more they stumble on one another, competing for the local resources in the client. When these small delays pile up one after the other, it becomes noticeable. It has been observed during the experiments that the CPU utilization of the client when running parallel models is higher, especially for high values of *m*.
- **Acknowledge Bottleneck:** When all the actors that run in parallel accomplish their tasks, they send a message back to the client for notification (as part of the RMI protocol). For our parallel model this happens at the same time due to the blocks having the same duration. This creates a second bottleneck. Therefore our parallel model with same block size for all blocks can be considered a worst case scenario for communication.
- **`AddSubstract` actor CPU and network Bottleneck:** The tree and especially the parallel topology experience and extra CPU load and bottleneck in the machine that runs the `AddSubstract` actor. This actor receives all the tokens produced by the Sleep actors and performs and addition on them. This makes this actor a special case because of two reasons. First of all, it represents a communication bottleneck since all the tokes produced by all actors connected to it have to go through and be processed through a serial network device. This could also be considered a worst case since those tokens are received at the same time. Second, the amount of computation it performs is a function of the number of actors connected to it.

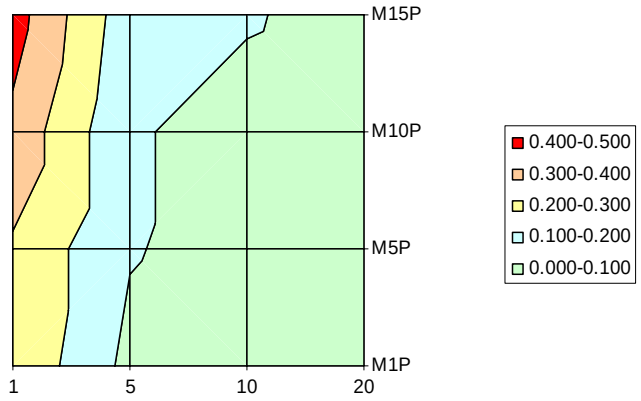


FIGURE 41: RELATIVE OVERHEAD FOR THE PARALLEL MODEL AND PARALLEL EXECUTION WITHOUT PIPELINING.

Despite the increase of overhead proportional to *m*, the speedup achieved is more significant and increasing the number of iterations can compensate it. An overhead of 10% is acceptable since it gives almost linear speedup.

In Figure 42, the relative overheads for the tree models are depicted. As expected, the result is in between that of the sequential and the parallel models. The relative overhead increases with *m* but to lesser extent. Here the bottlenecks are smaller due to the sizes of the intermediate parallel levels.

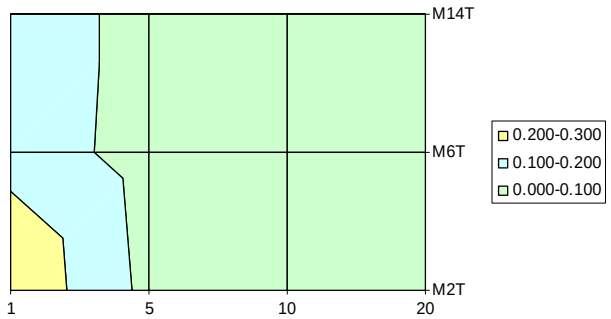


FIGURE 42: RELATIVE OVERHEAD FOR THE TREE MODEL AND PARALLEL EXECUTION WITHOUT PIPELINING.

The benefits that can be obtained from the distributed implementation are already evident with the results we have presented, except for the sequential case. In order to benefit from distributed simulations for the sequential case, we apply pipelining techniques in the next section.

6.3.3 PARALLEL EXECUTION WITH PIPELINING

Parallel execution with pipelining executes the parallel schedule using pipelining techniques. In this case, the schedule is the same as in the previous case, the difference is in the way we execute it.

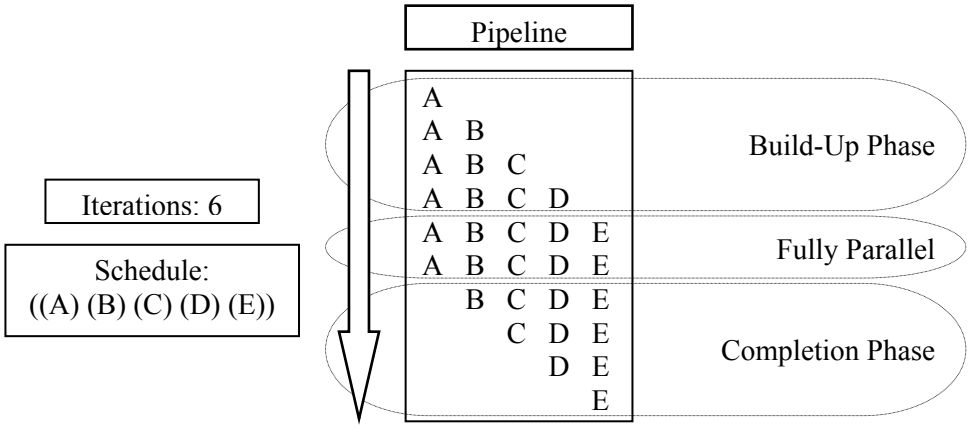


FIGURE 43: PIPELINING.

Figure 43 replicates Figure 9 in Section 4.11 to ease reading the following. It shows an example of a pipelined execution of a parallel schedule of a sequential model. First a build-up phase fills the queues of the actors with data. During this phase the schedule is executed from the beginning adding every time a new level. When including the last level, all the actors have enough tokens in their queues (as specified in their `TokenConsumptionRate` parameter) to be fired. From this point, a fully parallel phase can be executed. A completion phase commences after the first level has been executed as many times as the specified iterations. This empties the queues.

For models that contain cycles, since cycles are broken by including delayed tokens, it is important to notice that such models cannot benefit from the pipelined execution in general. Even though a sufficient number of delayed tokens can break the cycle and allow execution, this means the designer has to take this into consideration and this is against the objectives of this work.

At all times the queues contain a maximum of 2 times their `TokenConsumptionRate` parameter of the port. For this to happen and actor should receive the fire command, execute, and send the tokens over to another actor before the last has consumed his. This is something very exceptional.

With this dispatching model, the expected execution times are:

$$T_{seq}^{pp}[n,m] = ((n+m-1)*b) + \delta_{seq}^{pp}[n,m] \quad \textbf{(6-9)}$$

$$T_{par}^{pp}[n,m] = (n*b) + \delta_{par}^{pp}[n,m] \quad \textbf{(6-10)}$$

$$T_{tree}^{pp}[n,m] = ((n+\lceil \log_2(m+2) \rceil - 2)*b) + \delta_{tree}^{pp}[n,m] \quad \textbf{(6-11)}$$

$\delta^{pp}{}_{g}[n,m]$ represents various overheads that include among others, initialization, scheduling and communication in the distributed with pipelining case. For the sequential topology, the execution time is the sum of the number of iterations and the number of levels of the schedule minus 1. The levels of the schedule in this sequential case are the same as the number of blocks. This corresponds to the build-up phase as

in the example in Figure 43. For the parallel topology, the buffering phase has no impact on the result since all the Sleep actors can be executed in parallel. For the tree model, the build-up phase amounts to $\lceil log_2(m+2) \rceil - 2$. This corresponds to the number of levels in the schedule minus 1.

TABLE 4: PERFORMANCE RESULTS FOR THE PARALLEL EXECUTION WITH PIPELINING

Model	Makespan per nr. of iterations (s)			
	1	5	10	20
M1S	7	27	52	102
M5S	27	48	73	123
M10S	53	73	98	148
M15S	79	99	125	175
M1P	7	27	52	102
M5P	7	27	53	103
M10P	8	28	54	104
M15P	9	29	55	105
M2T	7	27	52	102
M6T	12	32	58	108
M14T	18	38	64	114

Table 4 shows the results for the parallel execution with pipelining. **All the models greatly benefit from the distributed simulation** as described in the equations (6-9) to (6-11). The parallel pipelined simulation achieves great speedups in all topologies, even for the ones that show a small degree of parallelism.

We calculate the relative overhead in relation with the overall makespan for this case. The processing is done in the following manner:

$$\rho_{seq}^{pp}[n,m] = \frac{T_{seq}^{pp}[n,m] - ((n+m-1)*b)}{T_{seq}^{pp}[n,m]} \quad \textbf{(6-12)}$$

$$\rho_{seq}^{pp}[n,m] = \frac{T_{par}^{pp}[n,m] - (n*b)}{T_{par}^{pp}[n,m]} \quad \textbf{(6-13)}$$

$$\rho_{seq}^{pp}[n,m] = \frac{T_{tree}^{pp}[n,m] - (n+(\lceil log_2(m+2) \rceil - 2)*b)}{T_{tree}^{pp}[n,m]} \quad \textbf{(6-14)}$$

The various overheads are shown in Figure 44, Figure 45 and Figure 46. The results are similar to the ones in the previous section. Again, all of them show low overheads for high values of *m* and *n*. Figure 44 shows the relative overheads for the sequential model. It is interesting to observe that the results are similar to the case without pipelining even though the makespan of the simulations are way smaller. This makes us consider higher traffic as explained in the section before a less dominant reason to explain the growth of overheads for the different cases.

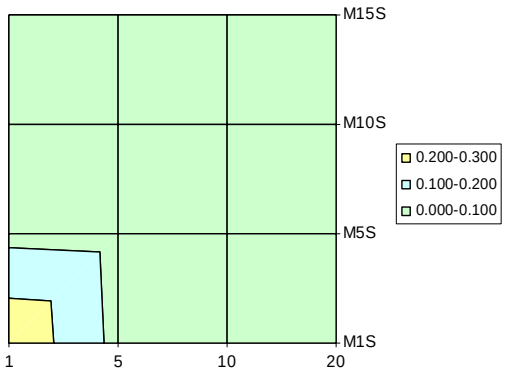


FIGURE 44: RELATIVE OVERHEAD FOR THE SEQUENTIAL MODEL AND PARALLEL EXECUTION WITH PIPELINING.

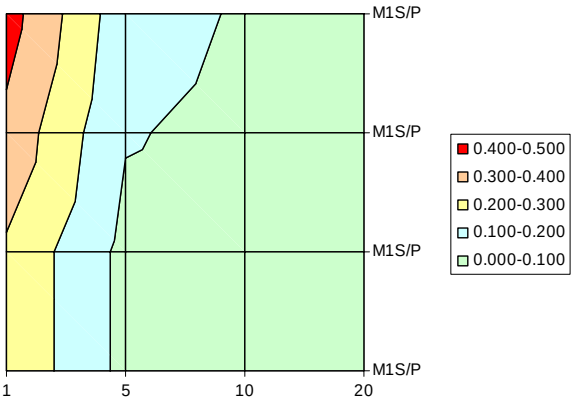


FIGURE 45: RELATIVE OVERHEAD FOR THE PARALLEL MODEL AND PARALLEL EXECUTION WITH PIPELINING.

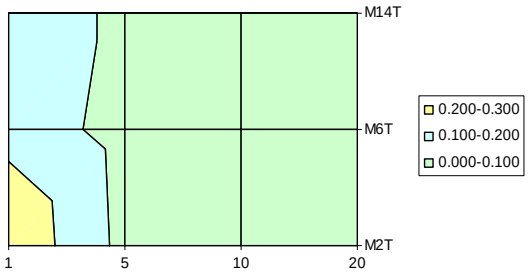


FIGURE 46: RELATIVE OVERHEAD FOR THE TREE MODEL AND PARALLEL EXECUTION WITH PIPELINING.

6.3.4 CONCLUSION ON TOPOLOGY AND NUMBER OF ACTORS

The main conclusion from these experiments is that the topology plays an important role when performing parallel simulations without pipelining. The lower the degree of parallelism of the topology, the less the benefit. To avoid this, we have implemented a pipelined version that successfully allows for all topologies to benefit from parallel simulation. This results in timing being greatly insensitive to the topology.

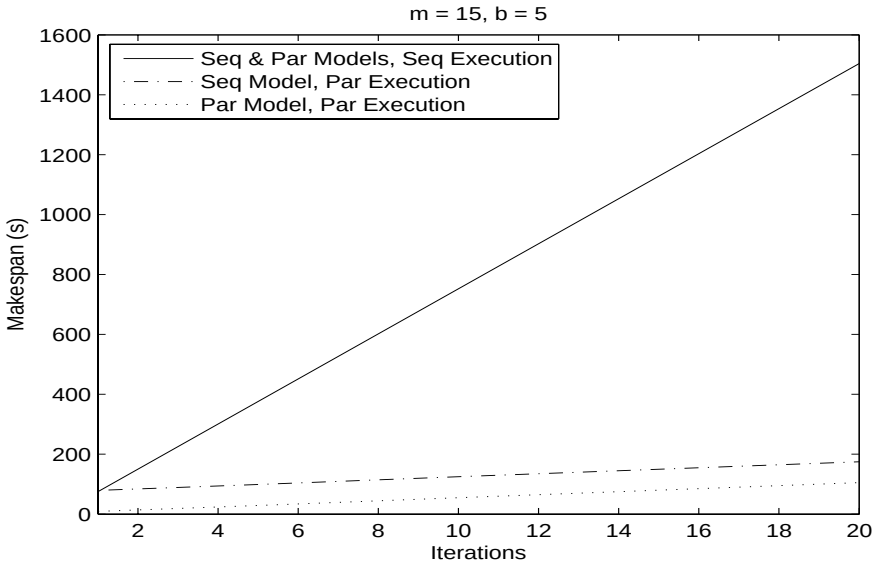


FIGURE 47: MAKESPAN COMPARISON, PIPELINED PARALLEL EXECUTION VS. THE EXISTING SEQUENTIAL IMPLEMENTATION.

Figure 47 shows the significant reduction of makespan achieved by the pipelined parallel execution as compared with the existing sequential implementation yielding linear speedup.

Relative overheads are largely insensitive to the number of blocks and number of iterations for large block size. In the next section we investigate effects of block size on the results.

6.4 BLOCK TIME AND NUMBER OF ITERATIONS

The only parameter we have not varied in the experiments so far is the `Sleep` actor block size. There has to be a limit to as how small the block size can be in order for the overheads to dominate the makespan. For this purpose, we have performed some experiments to find out where the limits are. Table 5 shows the results obtained for the 3 different topologies when executed both sequential and parallel. The number of actor has been fixed to 15 and the number of iterations to 20. All the sequential results are equal.

TABLE 5: PERFORMANCE RESULTS FOR DIFFERENT TIMES FOR THE SLEEP BLOCK.

Model	Makespan (s) for 20 iterations per Block Time (ms)				
	1	5	10	50	100
All Seq	1	2	3	15	30
M15S Par	3	4	4	6	7
M15T Par	3	4	4	5	6
M15P Par	4	4	4	5	6

We find that if we lower the block size down to around 14 ms, the sequential implementation is faster than the distributed one. In any case, the makespans at this level are so small that it does not really make a big difference between the 2 execution paradigms for a low number of iterations.

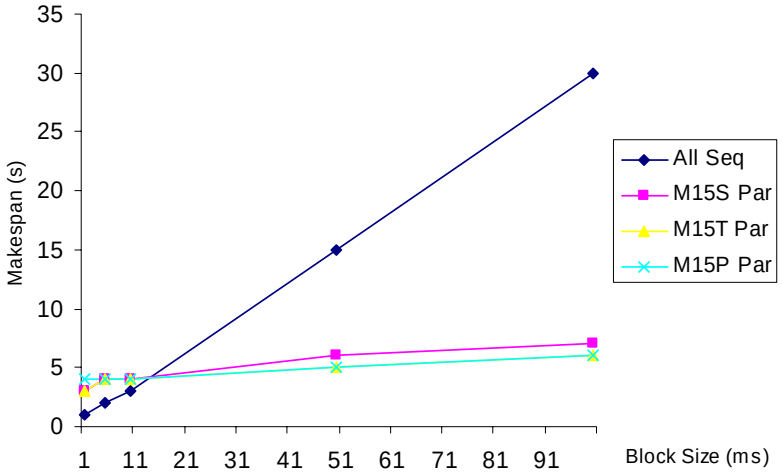


FIGURE 48: MAKESPAN FOR ALL THE TOPOLOGIES WITH 15 ACTORS, 20 ITERATIONS AND DIFFERENT BLOCK SIZES FOR SEQUENTIAL AND PARALLEL EXECUTION.

Since we have observed previously that a high number of iterations can compensate a small block size, we perform yet another experiment with all the 3 topologies, with 15 actors and we increase the number of iterations for both sequential and parallel execution.

TABLE 6: PERFORMANCE RESULTS FOR THE TOPOLOGIES WITH 5 MS BLOCK TIME AND DIFFERENT NUMBER OF ITERATIONS.

Model	Makespan (s) for 5 ms Block Time per nr. of iterations				
	50	100	500	1000	5000
All Seq	4	8	38	76	388
M15S Par	5	7	19	35	162
M15T Par	5	7	21	38	180
M15P Par	7	10	37	69	334

Table 6 shows the results the different topologies, 5 ms block size and high number of iterations. Even for a very small block time, with a high number of iterations, we can still yield better performance using the distributed approach. In this case, it is the sequential topology the one that benefits the most since it is the one less affected by communication overheads and bottlenecks.

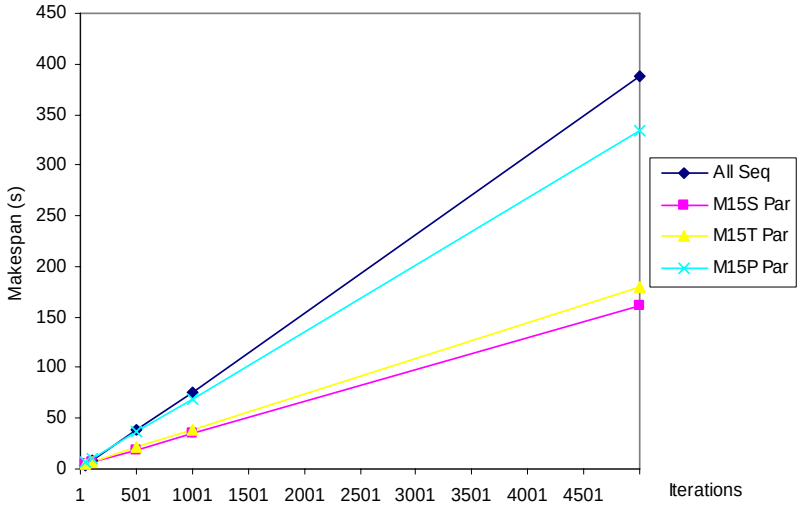


FIGURE 49: MAKESPAN FOR ALL TOPOLOGIES WITH 15 ACTORS, SEQUENTIAL AND PARALLEL EXECUTION AND HIGH NUMBER OF ITERATIONS.

It is important to notice, that with this small block size, the amount of computation performed by the `AddSubstract` actor starts becoming relevant in comparison. Figure 49 shows a graph were we can see how all the 3 topologies still benefit from a distributed simulation. The bigger the number of iterations, the more benefit. But there has to be a limit. We have chosen the best case, which is the sequential one and an extremely small block size for the next experiment, 1 ms. The sequential case does not contain actors that vary their computation with the number of actors connected to it. Table 7 shows the results for different number of iterations of the sequential model with 15 actors.

TABLE 7: PERFORMANCE RESULTS FOR THE M15S MODEL WITH 1 MS BLOCK TIME AND DIFFERENT NUMBER OF ITERATIONS.

M15S	Makespan for 1 ms Block Time per number of iterations (s)			
	50	100	500	1000
Seq	8	16	78	155
Par	15	30	138	272

All the results show worse performance for the parallel simulation in all cases. Increasing the number of iterations does not help so as expected there is a lower limit for the block size.

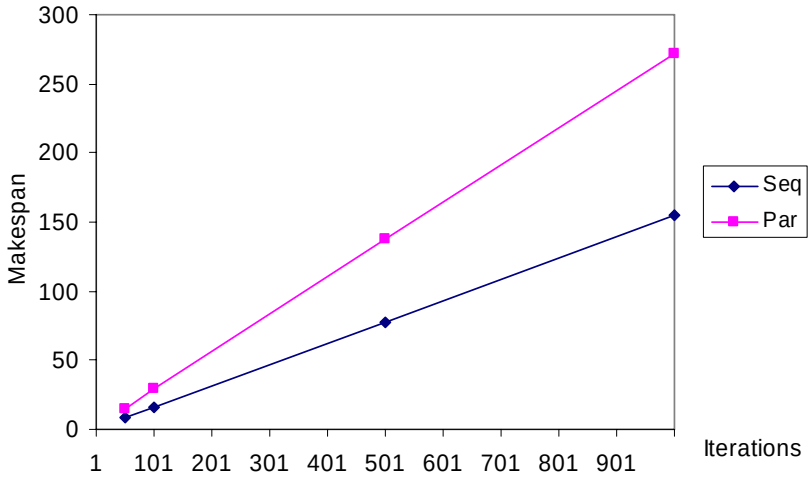


FIGURE 50: MAKESPAN FOR THE SEQUENTIAL MODEL WITH 15 ACTORS AND DIFFERENT NUMBER OF ITERATIONS.

Figure 50 depicts the results from Table 7. It can be observed that when reached a lower limit for the block size, increasing the number of iterations makes performance even worse.

6.4.1 CONCLUSION ON BLOCK SIZE AND NUMBER OF ITERATIONS

In conclusion, there is a lower limit to block size in order to benefit from parallel simulation. We find that limit to be very small, comparable to network latency. It is important to observe that our experiments are extreme cases, where all the actors have the same duration. Real models will have different block sizes for the different actors, and the performance for the simulation will be driven by the most expensive one. In case all the actors are very light, we can still benefit from distributed simulation by clustering two or more actors. This will result in a more expensive actor in terms of computation.

6.5 WHEN TO USE DISTRIBUTED EXECUTION

Now we are in the position to give some pointers on when to use distributed execution, based on the parameters investigated and described in Table 1. We recommend using pipelined parallel execution in general instead of the existing implementation. Equations (6-9) to (6-11) can be used to have a rather accurate estimation of the makespan before the simulation is performed. In the general case of having heterogeneous block sizes in the model, the biggest one is the one to use. On top of that, they can be compared with equation (6-1) in order to obtain an estimation of the speedup that can be obtained.

Parallel pipelined execution makes the results almost insensitive to the topology, allowing to greatly reducing execution times when simulating models with a low degree of parallelism.

There is mainly one scenario in which pipelined parallel execution yields worse results than the existing implementation. That occurs when we have extremely small block size *b* for the most expensive actor. We believe that hardly ever happens in real models. In any case, this scenario can be easily avoided by using clustering of actors, or simply merging two or more actors into one. Since the most expensive actor is the one dominating the simulation in terms of timing.

6.6 PLATFORM

For the sake of completeness, we describe the hardware platforms that have been used for the experiments. The computer that run the sequential experiments and acted as the client for the parallel experiments is described in Table 8. The computers that run the servers for the experiments are described in Table 9:

TABLE 8: CLIENT COMPUTER FEATURES.

Manufacturer	Fujitsu-Siemens
Model	S7110
Processor	Mobile Intel Pentium 4 – M, 2.2 GHz
Memory	1 Gb
OS	Windows XP, SP2
Netcard	100 Mb/s

TABLE 9: SERVER COMPUTERS FEATURES.

Manufacturer	HP
Model	Kayak
Processor	2 x 500 MHz Pentium III
Memory	512M + 2G swap
OS	Ubuntu Linux 5.04
Netcard	100 Mb/s

6.7 SUMMARY

The set of experiments to measure performance of the ADS implementation in Ptolemy II and the results have been presented. In the absence of available industrial case studies, we design experiments that explore the variability of relevant parameters, such as block size, number of blocks, number of iterations and topology. We study the effect of such variability on the overall makespan. Moreover, we derive an analytical expression for the expected results and check it against the empirical values obtained by the experiments. In this manner we can arrive to the relative overheads introduced by the ADS implementation as compared with the original implementation.

Results show important improvement over the original implementation of linear speedup. Topology of the model plays an important role highly affecting the results.

By using pipelined execution, the effect of topology is greatly decreased, but in general, models that contain cycles cannot benefit from it. As for the block size, there is a lower limit in order to benefit from parallel simulation, comparable to network latency. Small block size can be overcome with clustering.

We recommend using pipelined parallel execution in general instead of the existing implementation. Moreover, equations are provided to have a rather accurate estimation of the makespan beforehand. Speedups obtained by using ADS plus time saved in parallelization can help embedded systems design better tackle time to market (TTM) requirements.

Furthermore, in order to reduce traffic, the *vectorization* factor parameter can be used to reduce the number of synchronization barriers in the execution and the overall communication. We have not made experiments to measure the effects.

CHAPTER 7

CONCLUSIONS AND FURTHER WORK

Happiness is not a station you arrive
at, it is a way of traveling.
Anonymous

In this dissertation we propose Automated Distributed Simulation (ADS) that enables speeding up simulation by distributed computation. Since it is done transparently, it relieves programmers from the tedious and error-prone manual parallelization process such as dependency analysis. Two major challenges in embedded systems design are addressed: the use of abstractions and the implementation of new concepts in tools.

The classical untimed and sequential abstractions are not suitable for embedded systems which are intrinsically timed and distributed and require integration of computation and physicality. Abstractions such as the actor model and computational models used embedded systems modeling and design; provide the framework necessary for automating distributed simulations. We have chosen a tool that implements such abstractions for our work and extend it with our ideas. Different MoC and the implementation as domains, which are useful for embedded systems are described to motivate our choice. Furthermore, we identify the challenges and possibilities for distribution. Two different execution and communication mechanisms are analyzed and compared when moving to distributed environments. Distribution of time is an active area of research that presents two main approaches conservative and optimistic. None of them has proved to be applicable for the generality of systems thus an optimal general method may not exist. Among the different MoC we choose to implement SDF for ADS. SDF can be scheduled statically and as a communication mechanism distributed message passing is required. Different issues arise as identification of receivers in a distributed deployment. Fortunately, SDF does not include the notion of time. Moreover SDF is a popular formalism that is suited for a large number of applications and thus it is worthwhile to develop an efficient parallel implementation. Furthermore, the scheduling problem is introduced with the objective to propose a scheduling policy that minimizes the execution time of a simulation by using a distributed platform for the SDF MoC. Certain constraints have to be assumed including *zero inter-task communication times, full connectivity of PEs* and *availability of unlimited number of PEs* to simplify that allow for a polynomial time

algorithm. Moreover, a survey of previous scheduling work is presented, with emphasis on parallel scheduling within the Ptolemy project. Details about the current SDF scheduling policy are described and a parallel schedule computation technique is proposed that includes pipelining techniques for dispatching. This technique is optimal since it exploits both inherent parallelism of the model and the one arising from the iterative nature of SDF. We present the implemented framework that enables ADS of SDF Ptolemy models. It features correctness by preserving precedence constraints in the model. Optimality since it exploits all available parallelism in the models. Minimization of memory limitations, thus allowing bigger models. Automation of the distribution in a manner transparent to the designer. thus, avoiding the tedious and error prone task of parallelizing models. Client, server and service locator are described and how they constitute the distributed platform on which to run the simulations. The software that creates such platform has been described and implemented in relation with the existing software architecture. Peer discovery enables finding the required resources to run the simulation. Deployment of actors is optimized by sending MoML descriptions of the actors and loading the classes from local storage. A distributed message passing mechanism is implemented that overcomes identifying receivers at remote locations. The routing of data is decentralized for optimization. Finally a parallel dispatching and synchronization mechanism enables exploiting parallelism in the models in a correct manner. Pipelining techniques optimize simulation and enabling sequential models to benefit from ADS.

Finally, experiments are performed to measure the efficiency of simulations using ADS versus the original sequential implementation. Unfortunately, even though there are known industrial cases, for reasons of confidentiality they cannot be made available for performance tests thus we design experiments that explore the variability of relevant parameters, such as block size, number of blocks, number of iterations and topology. The different test cases explore inherent parallelism as well as the iterative nature of models as sources of parallelism. We provide expressions to compute the theoretical execution time and the relative overhead introduced by the implementation.

Results show important improvement over the original implementation of linear speedup. Topology of the model plays an important role that can be mitigated by using pipelined execution. A downside of pipelining is that it cannot be applied to the generality of cases; models that contain cycles are excluded. There is a lower limit for granularity that is comparable to network latency; luckily it can be overcome with clustering.

We recommend using pipelined parallel execution in general instead of the existing implementation. Speedups obtained by using ADS plus time saved in parallelization can help embedded systems design better tackle time-to-market requirements.

7.1 FUTURE WORK

Relevant future work includes experimentation with ADS and real life case studies to show examples of the benefits of ADS for industrial cases. Models that utilize Matlab are an interesting subject of study. The reduction in simulation time and the time spared by the automatic parallelization can speedup the time technologies take to reach the market.

Furthermore, clustering of traffic can be used in order to reduce traffic overheads the number of synchronization barriers. The *vectorization* factor parameter can be used for this purpose. We expect this to reduce the number of interactions between client and services and among services. On the other hand, high vectorization values might affect memory consumption due to data traffic in transit. Although considering the trade-offs, we believe this will increase efficiency.

Embedded systems are better modeled combining heterogeneous mixtures of MoC. Future work can extend the framework such as if the top model is SDF, every submodel governed by a different MoC can be run on a different machine. Obviously it would be more optimal to be able to partition those as well. Reutilizing common features to implement distributed versions of other domains completes the distributed simulation framework hereby proposed. Process Networks and Discrete Events are two interesting candidates.

PN could have an alternative implementation that puts aside synchronization steps. It would basically consist on a mechanism that triggers iteration automatically upon data availability. To avoid infinite sizes for queues, due to different speeds or computational requirements of the actors, a mechanism that implements queues on both sides of the communication can take care of pacing execution. This results in a distributed system with distributed control. For termination, a protocol can be implementented that requires minimal overhead. This can be a special token created in the system that would hold resposibility for deadlock detection (termination). Since the slowest process in the system is normally the one that paces execution, to minimize communication overhead, the token can flow through the system and find such an actor. Once the slowest actors queues are empty, it can trigger deadlock detection, if not detected, it can be passed on. Since PN is process based, a distributed version enables and eases experimentation with real concurrency.

DE is a popular formalism MoC for embedded systems that incorporates time. Due to the global notion of time it poses a challenge when moving to distributed environments. DDE addresses this issue incorporating a global notion of time and introducing extra overhead to keep local times updated. A truly distributed DDE implementation can benefit from the architecture described here. The drawback is that the conservative approach proposed in DDE cannot fully exploit inherent concurrency in the models since it avoids violating the causality constraint. An optimistic approach can fully exploit parallelism at the expense of rolling back modifications due to events

processed out of order. This can be achieved by using checkpointing [74]. A second benefit from using checkpointing is tacking reliability and fault tolerance [113] thus robustness of the system. This issue that has not been addressed in this work. Thus one effort can result in double benefit. This should be done in a way that avoids the domino effect [113] page 119. Checkpointing for both distributed time and fault tolerance are a complex ongoing research issues that can benefit fron the distributed infrastructure that is developed in this work.

Other improvements include parallelization of the initialization phase, allow remote loading of classes as opposed to local loading, feedback results, monitoring of servers, allow clustering, security, alternative discovery protocols (Jxta), alternative distributed interaction mechanisms (Corba), Matlab distributed simulation, hooks for tools (Uppaal).

BIBLIOGRAPHY

Modern technology
Owes ecology
An apology.
Alan M. Eddison

[1] Artist
Embedded Systems Design. The ARTIST Roadmap for Research and Development
Lecture Notes in Computer Science, Vol 3436.
`http://www.artist-embedded.org`

[2] F. Vahid, T. Givargis
Embedded System Design - A Unified Hardware/Software Introduction
John Wiley & Sons, Inc., 2002

[3] P. Lapsley, J. Bier, A. Shoham, E. A. Lee
DSP Processor fundamentals architectures and features
Wiley-IEEE Press 2001.

[4] S. A. Edwards
Languages for digital embedded systems
Kluwer Academic Publishers, 2000

[5] E. A. Lee
Embedded Software
Technical Memorandum UCB/ERL M01/26, University of California, Berkeley, 2001.

[6] J. Mitola III
Cognitive Radio: An Integrated Agent Architecture for Software Defined Radio.
Ph. D. Dissertation. Royal Institute of Technology, Sweden, 2000, ISSN 1403 – 5286.

[7] M. Katz, F. Fitzek
Cooperative Techniques and Principles Enabling Future 4G Wireless Networks
Computer as a Tool, EUROCON 2005 v. 1, p. 21-24.

[8] E. A. Lee
What are the key challenges in Embedded Software?
System Design Frontier, Volume 2, Number 1, January 2005
`http://www.hwswworld.com/`

[9] A. P. Black
Better Abstractions: an Agenda for Embedded Systems Research
Panel Presentation at DARPA-NSF Workshop on Embedded & Hybrid Systems

[10] E. A. Lee
What's Ahead for Embedded Software?
IEEE Computer, September 2000, p 18-26.

[11] M. V. Steen, A. S. Tanenbaum
Distributed Systems: Principles and Paradigms
Prentice Hall, 2002.

[12] S. Robinson
Simulation - The practice of model development and use.
Wiley 2004.

[13] G. A. Agha
ACTORS: A Model of Concurrent Computation in Distributed Systems
The MIT Press Series in Artificial Intelligence. MIT Press, Cambridge, 1986.

[14] E. A. Lee, S. Neuendorfer, M. J. Wirthlin
Actor-oriented design of embedded hardware and software systems
Journal of Circuits, Systems and Computers, 12(3): p:231-260, June 2003.

[15] C. Lopes, W. Hursch
Separation of Concerns
Northeastern University technical report NU-CCS-95-03, 1995.

[16] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, H. Zheng
Heterogeneous Concurrent Modeling and Design in Java
Memorandum UCB/ERL M05/21, EECS, UC Berkeley, 2005.

[17] D. Lazaro Cuadrado, A. P. Ravn, P. Koch
Automated Distributed Simulation in Ptolemy II
Parallel and Distributed Computing and Networks Proceedings, Acta Press 2007.

[18] J. Hernandez, P. de Miguel, M. Barrena, J. Martinez, A. Polo, M. Nieto
Parallel And Distributed Programming With An Actor-based Language
Parallel and Distributed Processing, 1994. Procs. 2nd Euromicro Workshop, 1994 p. 420-427.

[19] A. Marcoux, C. Maurel, P. Salle
AL1: a language for distributed applications
Distributed Computing Systems, 1988, Workshop on the Future Trends p: 270-276.

[20] Y. Zhao, T.H. Feng
A Framework for Distributed Modeling and Execution with Ptolemy II
`http://www.eecs.berkeley.edu/~tfeng/ee290proposal.html`

[21] The Kepler Project
Kepler Project
`http://kepler-project.org/`

[22] Department of EECS, UC Berkeley
Ptolemy II
`http://ptolemy.eecs.berkeley.edu/ptolemyII/`

[23] Department of EECS, UC Berkeley
Ptolemy Project
`http://ptolemy.eecs.berkeley.edu/`

[24] J. T. Buck, S. Ha, E. A. Lee, D. G. Messerschmitt
Ptolemy: A Mixed-Paradigm Simulation/Prototyping Platform in C++
International Journal of Computer Simulation, 4:155-182. April 1994.

[25] D. Harel, A. Pnuelli
On the development of reactive systems.
NATO Advanced Study Institute on Logics and Models for Verification and Specification of Concurrent Systems.
`Springer Verlag, 1985.`

[26] C. Brooks et. al
Heterogeneous Concurrent Modeling and Design in Java. Volume 1: Introduction to Ptolemy.
EECS Department, University of California, Berkeley, UCB/EECS-2007-7, 2007.

[27] C. Szyperski
Component Software, Beyond Object –Oriented Programming
`Addison-Wesley 1998.`

[28] C. Hewitt, P. Bishop, R. Steiger.
A Universal Modular Actor Formalism for Artificial Intelligence.
IJCAI 1973.

[29] G. Agha
Actors: A Model of Concurrent Computation in Distributed Systems
Doctoral Dissertation, MIT Press, 1986.

[30] E. A. Lee.
Embedded software.
Advances in Computers, 56, 2002.

[31] S. Neuendorffer
Actor-Oriented Metaprogramming
Ph.D. Thesis, University of California, Berkeley, 2004.

[32] E. A. Lee, E. Goei, H. Heine, W. Ho, S. Bhattacharyya, J. Bier, E. Guntvedt
Gabriel: A Design Environment for Programmable DSPs
Proc. of IEEE Design Automation Conf., pp. 141-146, Las Vegas, 1999.

[33] D. G. Messerschmitt
A Tool for Structured Functional Simulation
IEEE Proceedings, September 1987.

[34] S. Bhattacharyya et al.
The Almagest – Volume I, Ptolemy 0.7 User's Manual
University of California, Berkeley 1997

[35] A. Kalavade, E. A. Lee
Hardware/Software Co-Design Using Ptolemy - A Case Study
IFIP International Workshop on Hardware/Software Co-Design, Grassau, Germany, 1992

[36] K. P. Khiar, E. A. Lee
Modeling Radar Systems using Hierarchical Dataflow
IEEE Intl. Conf in Acoustics, Speech, and Signal Processing, p.3259-3262, USA, 1995.

[37] J. Liu, E. A. Lee
Timed Multitasking for Real-Time Embedded Software
IEEE Control System Magazine on Advances in Software Enabled Control 2003, p 65-75.

[38] P. Baldwin, S. Kohli, E. A. Lee
VisualSense: Visual Modeling for Wireless and Sensor Network Systems
Technical Memorandum UCB/ERL M04/08 University of California, Berkeley, USA, 2004.

[39] Y. Xiong and E. A. Lee
An Extensible Type System for Component-Based Design
Tools and Algorithms for the Construction and Analysis of Systems 2000, Berlin, Germany.

[40] C. Brooks et. al
Heterogeneous Concurrent Modeling and Design in Java. Volume 2: Ptolemy II Software Architecture.
EECS Department, University of California, Berkeley, UCB/EECS-2007-8, 2007

[41] C. Brooks et. al
Heterogeneous Concurrent Modeling and Design in Java. Volume 3: Ptolemy II Domains.
EECS Department, University of California, Berkeley, UCB/EECS-2007-9, 2007

[42] J. S. Davis II
Order and Containment in Concurrent System Design
PhD. Dissertation, University of California, Berkeley, 2000.

[43] E. A. Lee, D. G. Messerschmitt
Synchronous data flow
Proceedings of the IEEE, Vol. 75 No. 9. Pages 1235-1245, September 1987

[44] S. S. Bhattacharyya
Compiling Dataflow Programs for Digital Signal Processing
Ph.D. Dissertation, University of California at Berkeley, 1994.

[45] National Instruments
LabVIEW
http://www.ni.com/labview/

[46] E. A. Lee, D. G. Messerschmitt
Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing
IEEE Transactions on Computers, Vol. C-36 No. 1. Pages 24-35, January 1987.

[47] J. T. Buck.
Scheduling dynamic dataflow graphs with bounded memory using the token flow model.
Ph.D. Dissertation, Dept. of EECS, University of California, Berkeley, 1993.

[48] C. Cassandras, S. Lafortune
Introduction to Discrete Event Systems.
Kuwer Academic, 1999. ISBN 0-7923-8609-4.

[49] Edward A. Lee.
Modeling Concurrent Real-time Processes Using Discrete Events
Annals of Software Engineering, vol. 7 1999, p 25-45.

[50] Randy Brown.
CalendarQueue: A Fast Priority Queue Implementation for The Simulation Event Set Problem
Communications of the ACM, October 1998, Volume 31, Number 10.

[51] J. Misra.
Distributed Discrete-Event Simulation.
Computing Surveys, Vol. 18, No. 1, March 1986, pages 39-65.

[52] R. M. Fujimoto.
Parallel and Distributed Simulation Systems
Wiley 2000.

[53] G. Kahn.
The semantics of a simple language for parallel programming
Proceedings of the IFIP Congress 74 Amsterdam, The Netherlands, 1974.

[54] Khoral Inc.
Khoral
`http://www.khoral.com/`

[55] E. A. Lee, T. M. Parks
Dataflow Process Networks
Proceedings of the IEEE, v. 83, p. 773-799.

[56] T. M. Parks.
Bounded Scheduling of Process Networks
Ph.D. Dissertation. EECS Department, University of California. Berkeley, 1995.

[57] C.A.R. Hoare.
Communication Sequential Processes.
Prentica-Hall, 1985. Free electronic version from http://www.usingcsp.com/

[58] B. Roscoe, C.A.R. Hoare
The Laws of Occam Programming
Theoretical Computer Science 60, 1988, p. 177-229.

[59] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, A. Sangiovanni-Vincentelli
Hardware-software codesign of embedded systems.
IEEE Micro, Volume 14 Issue 4, August 1994, pages 26-36.

[60] A. Girault, B. Lee, E. A. Lee.

Hierarchical Finite State Machines with Multiple Concurrency Models.
IEEE Transactions On Computer-aided Design Of ICs And Systems, v. 18, n. 6, 1999.

[61] R. Alur, D. L. Dill
A Theory of Timed Automata
Theoretical Computer Science 126, p. 183-235, 1994

[62] G. Behrmann, A. David, K. G. Larsen
A Tutorial on Uppaal
SFM-RT 2004. LNCS 3185

[63] Thomas A. Henzinger, Benjamin Horowitz and Christoph Meyer Kirsch.
Giotto: A time-triggered language for embedded programming.
Technical Report UCB//CSD-00-1121,University of California,Berkeley,2000.

[64] A. Benveniste, G. Berry
The Synchronous Approach to Reactive and Real-Time Systems
Proceedings of the IEEE, v. 79, n.. 9, p. 1270-1282, 1991.

[65] S. A. Edwards
The Specification and Execution of Heterogeneous Synchronous Reactive Systems
Ph.D. thesis, University of California, Berkeley, May 1997. Available as UCB/ERL M97/31.

[66] B. Lee
Specification and Design of Reactive Systems
Ph.D. Dissertation, Memorandum UCB/ERL M00/29, University of California, Berkeley, 2000.

[67] P. Whitaker.
The Simulation of Synchronous Reactive Systems In Ptolemy II
Memorandum UCB/ERL M01/20, University of California, Berkeley, 2001.

[68] T. Amagbegnon, L. Besnard, P. Le Guernic.
Implementation of the Data-flow Synchronous Language Signal.
Programming Languages: Design and Implementation, ACM, 163-173, 1995.

[69] Gérard Berry.
The Foundations of Esterel.
Proof, Language, and Interaction: Essays in Honour of Robin Milner, G. Plotkin, C. Stirling, and M. Tofte, editors, MIT Press, Foundations of Computing Series, 2000.

[70] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud.
The Synchronous Dataflow Programming Language Lustre.
Proceedings of the IEEE, Vol. 79, No. 9, 1305-1320, September 1991.

[71] F. Maraninchi.
The Argos Language: Graphical Representation of Automata and Description of Reactive Systems.
Proceedings of the IEEE Workshop on Visual Languages, Kobe, Japan, October 1991.

[72] *A. G. Olson, B. L. Evans*
Deadlock Detection for Distributed Process Networks
Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Proc., 2005, vol. 5, pp. 73-76, USA.

[73] j. Zhou, T. Kuo-Chung
Deadlock analysis of synchronous message-passing programs
Software Engineering for Parallel and Distributed Systems, 1999, p. 62-69.

[74] T. H. Feng and Edward A. Lee
Incremental Checkpointing with Application to Distributed Discrete Event Simulation
Winter Simulation Conference (WSC 2006), Monterey, CA, December 3-6, 2006

[75] F. Cottet, J. Delacroix, C. Kaiser, Z. Mammeri
Scheduling in Real-Time Systems
Wiley 2002.

[76] R. Diestel

Graph Theory
Springer-Verlag, Electronic Edition, 2005.

[77] B. Parhami
Introduction to Parallel Processing, Algorithms and Architectures
Kluwer 2002.

[78] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, P. Wong.
Theory and practice in parallel job scheduling
Job Scheduling Strategies for Parallel Processing*, v. 1291, p. 1-34. LNCS Springer-Verlag, 1997.*

[79] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein
Introduction to Algorithms
The MIT Press, 2001.

[80] I. D. Scherson, L. M. Campos
Efficient Task Scheduling Heuristic for Multiprocessor Systems
Parallel and Distributed Computing and Systems, Las Vegas, USA, October 1998, pp 32-37.

[81] T. L. Casavant and J. G. Kuhl.
A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems.
Software Engineering, IEEE Transactions on, v.14, i.2, 1988, p. 141-154.

[82] S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi
Optimization by Simulated Annealing
Science, n. 4598, 1983

[83] D. A. Coley
An Introduction to Genetic Algorithms for Scientists and Engineers
World Scientific, 1997.

[84] D. Lázaro Cuadrado, C. Marcos Lagunar
Study of Parallelization of a Genetic Algorithm, used as a DSP Multiprocessor Scheduler, onto a Linux Cluster.
Master Thesis, Aalborg University (2000).

[85] Y. Kwok, I. Ahmad
Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors.
ACM Computer Surveys, Vol. 31, No. 4, December 1999.

[86] A. Gerasoulis, T. Yang
A comparison of clustering heuristics for scheduling DAGs on multiprocessors
Journal of Parallel and Distributed Computing*, v. 16, n. 4, p. 276-291, 1992.*

[87] B. Lee, A. R. Hurson
A vertically layered allocation scheme for dataflow systems
Journal of Parallel and Distributed Computing, v.1, p. 175-187, 1991.

[88] J. Bier, E. Goei, W. Ho, P. Lapsley, M. O'Reilly, G. Sih and E. A. Lee
Gabriel: A Design Environment for DSP
IEEE Micro Magazine, October 1990, vol. 10, no. 5, pp. 28-45

[89] E. A. Lee
Data Parallelism in Graphical Signal Flow Representations of Algorithms
Technical Report UCB/ERL M92/110, EECS Dept., University of California, Berkeley, CA 94720, August 13, 1992.

[90] J. L. Pino, S. Ha, E. A. Lee and J. T. Buck
Software Synthesis for DSP Using Ptolemy
Journal on VLSI Signal Processing, vol. 9, no. 1, pp. 7-21, Jan., 1995.

[91] G. C. Sih and E. A. Lee
Dynamic-Level Scheduling for Heterogeneous Processor Networks
Second IEEE Symposium on Parallel and Distributed Processing, December 1990.

[92] G. C. Sih and E. A. Lee
Declustering: A New Multiprocessor Scheduling Technique
IEEE Trans. on Parallel and Distributed Systems, vol. 4, no. 6, pp. 625-637, June 1993.

[93] P. K. Murthy and E. A. Lee
Optimal Blocking Factors for Blocked, Non-Overlapped Multiprocessor Schedules
Proc. of IEEE Asilomar Conf. on Signals, Systems, and Computers, 1994.

[94] J. L. Pino, S.S. Bhattacharyya and E. A. Lee
A Hierarchical Multiprocessor Scheduling Framework for Synchronous Dataflow Graphs
UCB/ERL M95/36, May 30, 1995.

[95] E. A. Lee, D. G. Messerschmitt
Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing.
IEEE Transactions on Computers, Vol. C-36, No. 1, January 1987.

[96] J.C. Huang, T. Leng
Generalized Loop-Unrolling: a Method for Program Speedup
IEEE Symposium on Application-Specific System and Software Engineering Technology (ASSET99)*, 1999, pp. 244-248*

[97] Hu, T.C.
Parallel sequencing and assembly line problems.
Oper. Res. 19, 6 (Nov.), 841–848, 1961.

[98] Fishburn, P. C.
Interval Orders and Interval Graphs.
John Wiley and Sons, Inc., New York, 1985.

[99] Coffman, E. G. and Graham, R. L.
Optimal scheduling for two-processor systems.
Acta Inf. 1, 200–213, 1972.

[100] Dror G. Feitelson.
Job Scheduling in Multiprogrammed Parallel Systems.
Technical Report IBM Research Report RC 19790 (87657), IBM T.J. Watson Research Center, Yorktown Heights, NY, October 1994. Extended Version.

[101] Behrooz Shirazi, Mingfang Wang, and Girish Pathak.
Analysis and Evaluation of Heuristic Methods for Static Task Scheduling.
J. Parallel and Distributed Computing, 10:222-232, 1990.

[102] B. A. Shirazi, A. R. Husson, K. M. Kavi.
Scheduling and Load Balancing in Parallel and Distributed Systems, chapter Introduction to Scheduling and Load Balancing.
IEEE Computer Society Press, Los Alamitos, CA, 1995. ISBN 0-8186-6587-4.

[103] Bokhari, S. H.
On the mapping problem.
`IEEE Trans .Comput. C-30, 5, 207–214, 1981.`

[104] Microsoft Research
AsmL
`http://research.microsoft.com/fse/AsmL/`

[105] D. Lázaro Cuadrado, P. Koch, A. P. Ravn
Using AsmL as an executable specification Language: SDF Schedulers in Ptolemy II
ASM 2004, May, 2004, Short Presentations Volume, p. 19-28.

[106] D. Lázaro Cuadrado, P. Koch, A. P. Ravn
AsmL Specification of a Ptolemy II Scheduler
LNCS, ASM 2003 Proceedings. Advances in theory and practice, March, 2003, Vol. 2589.

[107] Sun Microsystems
Java Remote Method Invocation Specification
Sun Microsystems 2004.

http://java.sun.com/j2se/1.5/pdf/rmi-spec-1.5.0.pdf

[108] Sun Microsystems
Jini Network Technology
http://wwws.sun.com/software/jini/

[109] J. D. Gradecki
Mastering JXTA, Building Java Peer-To-Peer Applications
Wiley 2002.

[110] P. Maymounkov, D. Mazieres
Kademlia: A Peer-To-Peer Information System Based on the XOR Metric
Proc. IPTPS'02, p. 53-65, 2002.

[111] K. Vanmechelen, J. Broeckhove
On using Jini and JXTA in lightweight grids
LNCS, 3470(2005), p. 384-393.

[112] R. Campbell
Managing AFS: The Andrew File System
Prentice Hall, 1998.

[113] A. Burns, A. Wellings
Real-Time Systems and Programming Languages
Addison-Wesley, 2001.

[114] H. R. Nielson, F. Nielson
Semantics with applications. A formal introduction.
Wiley 1992.

[115] S. N. Burris
Logic for Mathematics and Computer Science.
Prentice Hall 1998.

[116] M. Fitting
First-Order Logic and automated theorem proving.
Springer 1996.

[117] Y. Gurevich
Evolving Algebras 1993: Lipari Guide.
Specification and Validation Methods Ed. E. Börger, Oxford University Press, 1995, 9-36.

[118] M. Anlauff
Abstract State Machines: An introductory tutorial.
http://www.first.gmd.de/~ma/ppp/tutorial/index.htm

[119] R. Stärk, J. Schmid, E. Börger
Java and the Java Virtual Machine. Definition, Verification, Validation.
Springer, 2001.
http://www.inf.ethz.ch/~jbook/

[120] ASM Gofer
ASM Gofer
http://www.tydo.de/AsmGofer

[121] G. del Castillo
The ASM Workbench – A Tool Environment for Computer-Aided Analysis and Validation
LNCS v. 2031, p. 578-581, 2001.

[122] J. Visser
EvADE Tool
Master Thesis, Delft University, 1996.

[123] Michigan Interpreter
Michigan Interpreter
ftp://www.eecs.umich.edu/groups/gasm/interp2.tar.gz

[124] Montages
Montages
http://www.montages.com

[125] XASM: Executable ASM Language
XASM: Executable ASM Language
http://www.xasm.org

APPENDIX: ABSTRACT STATE MACHINES

A thesis relates mathematical and non-mathematical objects. It cannot have a mathematical proof, but it may be affirmed or negated by experimentation.
Yuri Gurevich

Ptolemy's component behavior is given by the programs of the simulator, so they must form the basis for any redesign. Ptolemy II is programmed in Java, a sequential programming language. Therefore, there is a need for a more abstract description. However, having a mathematical description on paper, separated from an implementation, is not attractive, because one would have to continually update it as the simulator evolves. Furthermore, one would still have to argue that there is a clear correspondence between the simulator program and the semantic definitions.

ASMs [117] (Abstract State Machines) have been successfully applied to give semantics to complex programming languages as Java [119]. AsmL (ASM Language) [104] is an executable software specification language based on ASMs. An AsmL implementation of Ptolemy components allows on one hand simulations and on the other reasoning about behavior. Moreover, it paves the way for generating parallel schedules in particular and for parallelizing the simulations in general. We have built two different AsmL specifications of the SDF scheduler. The first one generates a sequential schedule and models the scheduler implemented in Ptolemy II whereas the novel version generates parallel schedules, which was not previously implemented in Ptolemy II. The purpose of the first is to formalize the implementation to analyze it, whereas for the novel version it is to identify and make explicit parallelism in the Ptolemy II models. Moreover a parser has been developed in AsmL to allow reading of models created with Ptolemy II to test the specifications against the Java implementation in the tool. Figure 51 shows and overview of the study performed with AsmL.

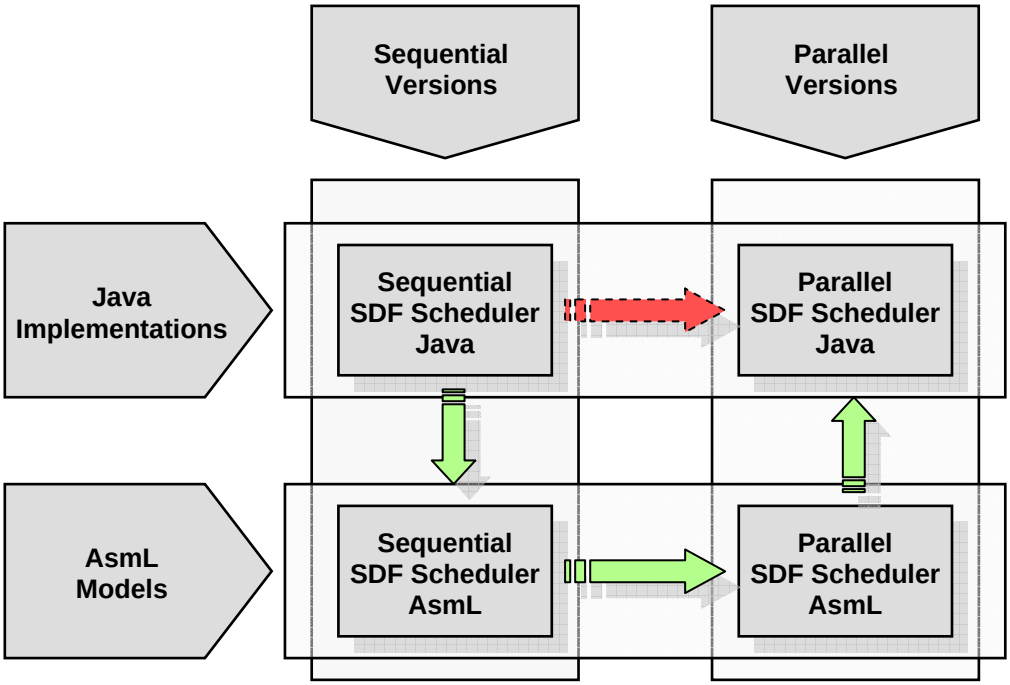


FIGURE 51: OVERVIEW OF THE USAGE OF ASML

Starting from a sequential SDF scheduler implemented in Java (existing in the Ptolemy II tool) we have created a parallel SDF scheduler in Java. But instead of performing this task directly we take two intermediate steps by specifying first the sequential scheduler in AsmL and from it creating an AsmL parallel scheduler. We want to emphasize that by sequential and parallel we mean that the resulting schedule calculated is either sequential or parallel.

The formal specification of the sequential SDF scheduler of Ptolemy II has proceeded in the following steps:

***Reverse Engineering*:** We used reverse engineering tools to understand the software architecture involved in the scheduling process. UML diagrams were generated from the Java code. The diversity of actors that can be part of Ptolemy models forces to some abstraction to deal with scheduling. The Actor interface provides this level of abstraction.

Analysis of the source code: The identified classes were manually inspected to extract the functionalities involved in the scheduling process. Starting from the _getSchedule method where the scheduling process is initiated, we analyzed the source code identifying the dependencies with other classes that were required.

***AsmL Programming*:** Once we had enough knowledge about the software architecture as well as the functionality implemented, we proceeded to implement the scheduler in AsmL. We used AsmL for Microsoft .NET 2.1.5.7.

We have developed a MoML parser allows interaction between the AsmL specification and Ptolemy II. MoML is a markup schema in XML (Extensible Markup

Language) used to store Ptolemy models in files. By being able to read the models we can make experiments with models created with Ptolemy II.

The AsmL models are available upon request by contacting the author.

The following is an overview of ASM for a better understanding of such models used in [105][106] .

A.1 ASMS

The Abstract State Machine (ASM) formalism (formerly called evolving algebras or ealgebras) was introduced by Yuri Gurevich [117] as an attempt to bridge the gap between formal models of computation and practical specification methods. It is a methodology based upon mathematics for describing simple abstract machines which correspond to algorithms. It provides the formal foundation for specification and verification of complex dynamic systems. It is problem-oriented and a user-friendly specification method. Although ASMs can be seen merely as specification formalism, strictly speaking, ASMs constitute a computation model on structures (static algebras). ASMs offer a framework for highly abstract system description and analysis, model refinement allowing for functional correctness proofs.

A.2 ASMS FOR SYSTEM SPECIFICATION

System design is typically organized as series of refinement steps: starting from a high level description of the system, different steps refine intermediate description stages towards a low-level description. This is called the top-down design approach. Ideally, the last description stage is close to the implementation. The ASM method proposes to describe each stage of the refinement process in terms of ASMs.

A.3 ASM VS. TURING MACHINES

ASMs provide operational semantics (See Section A.15) for algorithms by elaborating upon Turing's thesis (Every algorithm is simulated by an appropriate Turing machine). We could declare a Turing machine to be the meaning of a program. But a Turing machine may require a long sequence of steps to simulate one step of an algorithm. For example, some programming languages can perform a matrix multiplication in one step, while a Turing machine will take a lot. There is a huge difference in the abstraction level. Simulation should be lock-step, taking only a bounded number of steps to simulate one step of the given algorithm.

The program of an ASM is like the program of a Turing machine - a description of how to modify the current configuration of a machine in order to obtain a possible successor configuration. The main difference between an ASM and a Turing machine is that the configurations of an ASM are mathematical structures rather than strings. (The tape of a Turing machine can be seen as a string. Every Turing machine can be viewed a particularly simple ASM.) ASMs perform computations on structures in the

sense that they obtain a structure as input, modify this structure step by step, and output the resulting structure when reaching a halting state.

The fact that ASMs work on structures rather than strings has important consequences: the ASM computation model is, in some sense, more powerful and universal than the standard computation models in theoretical computer science.

A.4 ASM CHARACTERISTICS

The ASM methodology has the following desirable characteristics

- **Precision:** One uses a specification methodology to describe a system by means of a particular syntax and associated semantics. If the semantics of the specification methodology is unclear, descriptions using the methodology may be no clearer than the original systems being described. ASMs use classical mathematical structures to describe states of a computation; structures are well-understood, precise models. This distinguishes ASMs from informal methodologies.
- **Faithfulness:** Given a specification, how does one know that the specification accurately describes the corresponding real system? Since there is no method in principle to translate from the concrete world into an abstract specification, one needs to be able to see the correspondence between specification and reality directly, by inspection. ASMs allow for the use of the terms and concepts of the problem domain immediately, with a minimum of notational coding. Many popular specification methods require a fair amount of notational coding which makes this task more difficult.
- **Understandability:** How easy is it to read and write specifications using a particular methodology? If the system is difficult to read and write, few people will use it. ASM programs use an extremely simple syntax, which can be read even by novices as a form of pseudo-code. Other specification methods, notably denotational semantics, use complicated syntax whose semantics are more difficult to read and write.
- **Executability:** Another way to determine the correctness of a specification is to execute the specification directly. A specification methodology which is executable allows one to test for errors in the specification. Additionally, testing can help one to verify the correctness of a system by experimenting with various safety or liveness properties. ASM specification can be executed directly. This allows one to test for errors in the specification. This can help one to verify the correctness. Methods such as VDM, Z, or process algebras are not directly executable.
- **Scalability:** ASM allow to describe a system at several different layers of abstraction. With multiple layers, one can examine particular features of a system while easily ignoring others. Proving properties about systems also can

be made easier, as the highest abstraction level is often easily proved correct and each lower abstraction level need only be proven correct with respect to the previous level. Some specification methodologies, *e.g.* Knuth's MIX language, provide only a single level of abstraction.

- **Generality:** ASMs are useful in a wide variety of domains: sequential, parallel, and distributed systems; abstract-time and real-time systems; finite-state and infinite-state domains. Many methodologies (*e.g.* finite model checking, timed input-output automata, various temporal logics) have shown their usefulness in particular domains; ASMs have been shown to be useful in all of these domains.

A.5 THE ASM THESIS

The ASM thesis is that any algorithm can be modeled at its natural abstraction level by an appropriate ASM.

A.6 HOW DOES AN ASM SIMULATE AN ALGORITHM?

ASMs model the behaviour of discrete dynamic systems in terms of sequences of transitions where structures are used as abstract representations of system states.

- The algorithm starts in some initial state S_0 and goes through states S_1, S_2, etc.
- A state is represented by a *static algebra*.
- We get from S_i to S_{i+1} by performing a bounded number of steps is S_i.
- Theses steps can be formulated as *transition rules*.

First of all, abstract states (static algebras) are described followed by transitions rules.

A.7 (ABSTRACT) STATES

ASMs are close to logic. They combine declarative concepts of first-order logic with the abstract operational view of transition systems in a unifying framework for mathematical modeling of computer-based systems. Each state can be represented by a first-order logic structure, a set with relations and functions. Some formal definitions are provided for a better understanding of the terms used.

A.7.1 FIRST-ORDER LOGIC

First-order logic is the simplest logic that allows us to easily and naturally express a truly wide range of important concepts and assertions in mathematics [75]. For computer scientists, the first order-logic is often a convenient intermediate stage in the translation of a mathematical problem into the clause logic.

First-order logic is the simplest logic that allows us to easily and naturally express a truly wide range of important concepts and assertions in mathematics [75]. For

computer scientists, the first order-logic is often a convenient intermediate stage in the translation of a mathematical problem into the clause logic.

First-order logic is "classical" logic. Each sentence is true or false. The new ingredient in first order logic is the use of the quantifiers $\exists$ and $\forall$. There is a unary negation operator and standard connectives are used: conjunction, disjunction and implication. Determining whether or not a first-order theory is consistent is in general semi-decidable. This means that there are algorithms that are guaranteed to terminate with the answer "yes" if the theory is consistent, but might never terminate if the theory is inconsistent. There can be no algorithm that will always terminate in both cases.

DEFINITION A-1: FIRST-ORDER LANGUAGE

A first-order language *L(R, F, C)* is determined by specifying:

- A set *R* of relation symbols *(f, g, h...)* with associated arities.
- A set *F* of function symbols *(r, r1, r2...)* with associated arities.
- A set *C* of constant symbols *(c, d, e...)*.

We must define what domain quantifiers quantify over and how we are interpreting constants, functions and relations with respect to that domain. These two items specify a model, also called structure.

DEFINITION A-2: STRUCTURE

A structure for a first order language *L(R, F, C)* is a pair *T = (D, I)* where:

- *D* is a nonempty set, called the domain of *S*.
- I is an interpretation.

DEFINITION A-3: INTERPRETATION

An interpretation I is a mapping of the first order language *L* on a domain *D*:

- $I(c)$ is an element of *D* for each constant symbol *c* in *C*.
- $I(f)$ is an *n*-ary function on *D* for each *n*-ary function symbol f in *F*.
- $I(r)$ is an *n*-ary relation on *D* for each *n*-ary relation symbol *r* in *R*.

Formerly, ASMs were called evolving algebras. Since states are represented by structures, they should have been called evolving structures, but for technical reasons, relations were replaced by appropriate functions. In the branch of mathematics called universal algebra, a first order structure without relations is called algebra. Therefore, the term “static algebra” is used to denote state.

A.8 STATIC ALGEBRAS (STATES)

In an ASM state, data come as abstract elements of domains (called universes) together with basic operations represented by functions. Domains are defined on the union of all domains (called superuniverse). Static Algebras represent *snapshots* of the computational processes. They can be seen as memory.

DEFINITION A-4: VOCABULARY OR SIGNATURE

A vocabulary Σ is a finite collection of function names, each of a fixed arity. The arity of a function name is the number of arguments the function takes. Function names can be static or dynamic.

- Relations are treated as boolean valued functions.
- Nullary function names are often called constants.
- For dynamic nullary functions the interpretation can change from one state to the next, so they correspond to variables.

Every ASM vocabulary contains the following functions (seen as function symbols):

- The equality sign (first order logic with equality).
- Nullary names *true*, *false*, *undef* and unary name *Boole.*
- Names of the usual boolean operations.

With exception of *undef*, all these logic names are relational. The constant *undef* represents an undetermined object, the default value of the superuniverse. It is used to allow partial functions.

The vocabulary should not change with the evolution of the machine. It reflects the invariant part of the algorithm.

A.9 FUNCTION TYPES

The basic functions of a static algebra can be categorized as static, dynamic and external.

A.9.1 STATIC FUNCTIONS

Static Functions are those basic functions that cannot be updated during the run of an ASM.

A.9.2 DYNAMIC FUNCTIONS

Dynamic Functions are those basic functions that may be updated during the run of an ASM.

A.9.3 EXTERNAL FUNCTIONS

External Functions are used to model behavior from the outside world, e.g. user input, operating system activities… External functions cannot be changed by rules of the ASM. They may be different in different states of the ASM.

An external function is an oracle, an unpredictable black box that is used but not controlled by the ASM.

EXAMPLE A-1

The vocabulary Σ_{bool} of Boolean algebras contains:

- two constants *0* and *1*
- a unary function name "-"
- two binary function names "+" and "*"

DEFINITION A-5: STATIC ALGEBRA OR STATE

A state S of the vocabulary Σ is a non-empty set X (called the superuniverse of Σ), together with interpretations of the function names in Σ on X.

- If f is an n-ary function of Σ, its interpretation is $f^S : X^n \to X$.
- If c is a constant of Σ, its interpretation c^S is an element of X.

The superuniverse X of the state S is denoted by $|S|$. Interpretations are also mentioned as locations in the literature.

EXAMPLE A-2

In Example A-1, the superuniverse of the state S is the set *{0, 1}*. The functions are interpreted as follows, where a, b are *0* or *1*

$0^S := 0$	*(zero)*
$1^S := 1$	*(one)*
$\neg^S := 1 - a$	*(logical complement)*
$a +^S b := \max(a, b)$	*(logical or)*
$a *^S b := \min(a, b)$	*(logical and)*

The state S is so-called a Boolean algebra.

DEFINITION 7-6: TERM

Terms are defined by induction. The terms of Σ are syntactic expressions generated as follows:

- Variables $v_0, v_1, ...$ are terms.
- Constants c of Σ are terms.
- If f is an n-ary function name of Σ and $t_1, ..., t_n$ are terms, then $f(t_1, ..., t_n)$ is a term.

EXAMPLE A-3

The following are terms of the vocabulary Σ_{bool}:

$$v_0 + v_1 \qquad 1 + (v_7 * 0)$$

Since terms are syntactic objects, they do not have a meaning. A term can be evaluated in a state, if elements of the superuniverse are assigned to the variables of the term.

DEFINITION A-7: VARIABLE ASSIGNMENT

Let S be a state. A variable assignment for S is a function A which assigns to each variable v_i an element $A(v_i) \in |S|$. Assignment of the element a to the variable x is written as follows:

$$A_x^a(v_i) = \begin{cases} a, & \text{if } v_i = x; \\ A(v_i), & \text{otherwise} \end{cases}$$

DEFINITION A-8: INTERPRETATION OF TERMS

Let S be a state of Σ, A be a variable assignmet for S and t be a term of Σ. The interpretation of term t under S and A $I(t)_A^S$ is defined as follows:

1. $I(v_i)_A^S := A(v_i)$
2. $I(c)_A^S := c^S$
3. $I(f(t_1,...,t_n))_A^S := f^S(I(t_1)_A^S,...,I(t_n)_A^S)$

EXAMPLE A-4

Consider the state S for Σ_{bool}. Let A be a variable assignment with $A(v_0) = 0$, $A(v_1) = 1$ and $A(v_2) = 1$. Then we have:

$$I((v_0 + v_1) * v_2)_A^S = 1$$

DEFINITION A-9: FORMULA

Let Σ be a vocabulary. The formulas of Σ are generated as follows:

- If s and t are terms of Σ, then $s = t$ is a formula.
- If F is a formula, then $\neg F$ is a formula.
- If F and G are formulas, then $(F \wedge G)$, $(F \vee G)$ and $(F \rightarrow G)$ are formulas.
- If F is a formula and x a variable, then $(\forall x F)$ and $(\exists x F)$ are formulas.

A formula $s = t$ is an equation. The expression $s \neq t$ is an abbreviation of the formula $\neg(s = t)$.

DEFINITION A-10: INTERPRETATION OF FORMULAS

Let S be a state of Σ, F be a formula and A be a variable assignment in S. The interpretation of a formula $I(F)_A^S \in \{True, False\}$ is defined as follows:

$$I(s=t)_A^S := \begin{cases} True, & if\ I(s)_A^S = I(t)_A^S; \\ False, & otherwise. \end{cases}$$

$$I(\neg F)_A^S := \begin{cases} True, & if\ I(F)_A^S = False; \\ False, & otherwise. \end{cases}$$

$$I(F \wedge G)_A^S := \begin{cases} True, & if\ I(F)_A^S = True\ and\ I(F)_A^S = True; \\ False, & otherwise. \end{cases}$$

$$I(F \vee G)_A^S := \begin{cases} True, & if\ I(F)_A^S = True\ or\ I(F)_A^S = True; \\ False, & otherwise. \end{cases}$$

$$I(F \rightarrow G)_A^S := \begin{cases} True, & if\ I(F)_A^S = False\ or\ I(F)_A^S = True; \\ False, & otherwise. \end{cases}$$

$$I(\forall x F)_A^S := \begin{cases} True, & if\ I(F)_{A_x^a}^S = True\ for\ all\ a \in S; \\ False, & otherwise. \end{cases}$$

$$I(\exists x F)_A^S := \begin{cases} True, & if\ I(F)_{A_x^a}^S = True\ for\ all\ a \in S; \\ False, & otherwise. \end{cases}$$

A.10 TRANSITION RULES

Transition rules are responsible for transforming one abstract state to another. Subsequent application of transition rules define a notion of execution where the abstract states are given as static algebras (That is why the original name of ASMs has been "Evolving Algebras"). They evolve by being updated during computations. Updating abstract states means to change the interpretation of some of the functions in the vocabulary. Transition rules are atomic actions in the sense that they perform bounded work (bounded inspection + bounded change).

DEFINITION A-11: TRANSITION RULES

Let Σ be a vocabulary. The transition rules P, Q of an ASM are syntactic expressions generated as follows:

1. Skip Rule:

```
skip
```

Meaning: Do nothing. The state before and after the execution remains the same.

2. Update Rule:

$$f(t_1,...,t_n) := s$$

Syntactic conditions:

- f is an n-ary dynamic function name of Σ
- $t_1,...,t_n$ and s are terms of Σ

Meaning: In the next state, the value of the function f at the arguments $t_1,...t_n$ is updated to s. It is allowed that f is a 0-ary function, i.e., a constant. In this case, the update has the form $c:=s$.

3. Block Rule:

```
[do in-parallel]
    R1
    R2
[enddo]
```

Meaning: R_1 and R_2 are executed in parallel. The resulting state is the union of the states obtained as if we executed R_1 and R_2 separately.

4. Conditional Rule:

if F **then** R_1 **else** R_2

Meaning: If F (the guard formula) is true, then execute R_1, otherwise execute R_2. The resulting state is as if we execute R_1 (if F is true) or as we execute R_1 (if F is false).

5. Let Rule:

let $x = t$ **in** P

Meaning: Assign the value of t to x and execute P. In the next state, the value of the variable x is updated to t.

6. Forall Rule:

forall x with F do P

Meaning: Execute P in parallel for each x satisfying F. The resulting state is the union of all the resulting states obtained of applying R to all x satisfying F.

7. Call Rule:

$r(t_1,...,t_n)$

Meaning: Call r with parameters $t_1,...t_n$. State before and after is the same.

A rule definition for a rule name r of arity n is an expression

$r(x_1,...,x_n) = R$

A.11 ABSTRACT STATE MACHINES

The notion of Abstract State Machines (ASMs) captures in mathematically rigorous yet transparent form some fundamental operational intuitions of computing, and the notation is familiar with programming practice and mathematical standards.

This allows to work with ASMs without any further explanation, viewing them as *pseudocode over abstract data*.

ASMs are systems of finitely many transition rules of form:

`if` *Condition* **`then`** *Updates*

which transform abstract states.

Condition: Also called guard, is an arbitrary first-order formula without free variables. If it evaluates to true then *Updates* are performed.

Updates: is a finite set of function updates (containing only variable free terms) of form:

$$f(t_1,...,t_n) := t$$

whose execution is to be understood as changing (of defining, if there was none) the value of the (location presented by the) function *f* at the given parameters.

DEFINITION A-12: ASM

An *abstract state machine* M consists of a vocabulary Σ, and initial state S_0 for Σ, a rule definition for each rule name, and a distinguished rule name of arity zero called the *main rule name* of the machine.

The semantics of transition rules is given by sets of updates. Since due to the parallelism (in the Block and the Forall rules), a transition rule may prescribe to update the same function at the same arguments several times, we require such updates to be consistent.

DEFINITION A-13: UPDATE

An update for *S* is a triple $\left(f,(a_1,...,a_n),b\right)$, where *f* is an *n*-ary dynamic function name, and $a_1,...,a_n$ and *b* are elements of |*S*|.

A function update assigns a new value to a function at the specified position. The meaning of the update is that the interpretation of the function *f* in *S* has to be changed at the arguments $a_1,...,a_n$ to the value *b*. The pair of the first two components of an update is called a location. An update specifies how the function table of a dynamic function has to be updated at the corresponding location. An update set is a set of updates.

DEFINITION A-14: CONSISTENT UPDATE SET

An update set U is called consistent, if it satisfies the following property:

$$If\ \left(f,(a_1,...,a_n),b\right) \in U\ and\ \left(f,(a_1,...,a_n),c\right) \in U, then\ b = c.$$

That means that a consistent update set contains for each function and each argument tuple at most one value.

DEFINITION A-15: FIRING OF UPDATES

The result of firing a consistent update set U in a state S is a new state S' with the same superuniverse as S satisfying the following two conditions for the interpretations of function names f of Σ.

1. If $\left(f,\left(a_1,...,a_n\right),b\right)\in U$, then $f^{s'}\left(a_1,...,a_n\right)=b$.
2. If there is no b with $\left(f,\left(a_1,...,a_n\right),b\right)\in U$ and f is not a monitored function, then $f^{s'}\left(a_1,...,a_n\right)=f^{s}\left(a_1,...,a_n\right)$.

All updates that are performed in a state S are executed in parallel. There is no memory about the updates performed previously, only the current state S_i. Inconsistent updates cause the computation to halt (e.g. a:= true, a:= false).

DEFINITION A-16: RUN OF AN ASM

Let M be an ASM with vocabulary Σ, initial state S and a main rule name r. Let A be an assignment. A run is a finite or infinite sequence $S_0,S_1,...$ of states for Σ such that the following conditions are satisfied:

1. $S_0=S$.
2. If $I\left(r\right)_A^{S_n}$ is not defined of inconsistent, then S_n is the last state in the sequence.
3. Otherwise, S_{n+1} is the result of firing $I\left(r\right)_A^{S_n}$ in S_n.

A.12 APPLICATIONS

The method built around the notion of Abstract State Machine (ASM) has been proved to be a scientifically well founded and an industrially viable method for the design and analysis of complex systems, which has been applied successfully to programming languages, protocols, embedded systems, architectures, requirements engineering, etc.

Plentiful examples exist in the literature of ASMs applied to different types of algorithms. Of particular interest in an industrial or scientific system design context are concurrent and embedded systems. Typical examples are distributed control systems.

Here some of the applications are mentioned:

- Abstract Algorithms
- Architectures (hardware and software)
- Benchmark Examples
- Compiler Correctness
- Databases
- Distributed Systems
- Hardware
- Industrial Reports

- Java
- Logic & Computability
- Mechanical Verification
- (other) Models of Computation
- Montages
- Natural Languages
- Programming Languages
- Protocols
- Real-Time Systems
- Security
- Verification
- VHDL
- WAM/Logic Programming

A.13 AVAILABLE TOOLS

Analysis with ASMs covers both verification and validation, using mathematical reasoning or experimental simulation (by running the executable models). For experimental simulation, various tools are available:

ASM Gofer [120] is an advanced ASM programming system, which runs under several platforms (Unix-based or MS-based operating systems). The aim of the AsmGofer system is to provide a modern ASM interpreter embedded in the well known functional programming language Gofer. Gofer is a subset of Haskell, the de-facto standard for strongly typed lazy functional programming languages.

The ASM Workbench [121] is a tool environment for supporting the design and validation of ASM specifications. The ASM Workbench is based on a specification language called ASM-SL, which includes a typed version of the language of abstract state machines (as defined by Gurevich) and additional constructs to define the underlying data model with the help of generic data types (like lists, finite sets, finite maps, etc.)

EvADE [122] is a tool that comprises both a compiler and a run-anlyzer for ASM. It supports dynamic sorts and non-determinism, and it accepts many-sorted ASMs with functional modules.

GEM-MEX is the support environment for the Montages method and stands for "Graphical Editor for Montages and Montages Executable Generator". Gem-Mex guides the user through the process of designing and specifying a language and generates a specialized programming environment which allows to analyze, execute,

and animate programs of the specified languages. The language designer can use the generated environment as a semantics inspection tool to test and validate design decisions.

Michigan Interpreter [123] is an ASM interpreter.

Montages [124] is a version of ASMs specifically tailored for specifying the static and dynamic semantics of programming languages. Montages combine graphical and textual elements to yield specifications similar in structure, length, and complexity to those in common language manuals, but with a formal semantics.

Object-based Mapping Automata: ASM are an attempt at a formal and operational specification method that allows to specify a system at various levels of abstraction. Object-based Mapping Automata (OMA) simplify basic ASM constructions and at the same time add a few concepts that support an object-based view of specifications.

XASM [125] is an executable ASM Language. XASM stands for Extensible ASM, realizes a component-based modularization concept based on the notion of external functions as defined in ASMs.

AsmL [104] is the Abstract State Machine Language. It is an executable specification language based on the theory of ASMs It uses XML and Word for literate specifications. It generates C++. AsmL is useful in any situation where you need a precise, non-ambiguous way to specify a computer system, either software or hardware. AsmL specifications are an ideal way for teams to communicate design decisions. Program managers, developers, and testers can all use an AsmL specification to achieve a single, unified understanding. One of the greatest benefits of an AsmL specification is that you can execute it. That means it is useful before you commit yourself to coding the entire system. An AsmL specification is also useful after you have refined your AsmL specification to an implementation (either by formal or informal means). It is important to make sure that the specification and implementation agree. A first prototype of AsmL has been used for the modeling, analysis and rapid prototyping of Universal Plug and Play devices and protocols. It was also used to model and perform conformance checking of the network configuration mechanism for the next incarnation of Windows 2000.

A.14 EXAMPLES

In this section some examples are given in order to illustrate the usage of ASMs.

A.14.1 A STACK MACHINE (REVERSE POLISH NOTATION)

Consider a stack machine that computes expressions given in reverse Polish notation.

$$(1+23) * (45+6) \Rightarrow 1\ 23 + 45\ 6 + *$$

Pseudo Code

The pseudo-code of the machine can be written as follows:

```
Algorithm (stack machine):
  reads the next entry from the RPN expression
       if it is a datum → push onto the stack
       if it is an operation →
            pops two items from the stack
            applies the operation to them
```

Universes (Domains):

The following domains are proposed for the stack machine:

- **Data:** e.g. integers
- **Oper:** e.g. {+, *}
- **List:** all lists composed of Data and Oper
- **Stack:** all stacks of Data

Functions:

The following functions can be defined:

- **Apply:** Oper × Data × Data → Data

For a simplified notation, we can consider that Apply(f, x, y) = f(x, y). Therefore, for the common operations of a stack machine we can define:

- **Push:** Data × Stack → Stack
- **Pop:** Stack → Stack
- **Head:** List -> {Data ∪ Oper}
- **Tail:** List → List
- **S:** Stack (distinguished element, initially = empty)
- **L:** List (distinguished element, initially = input expr.)
- **Arg1, Arg2:** Data (initially = undef)

Program of the ASM:

The program consists of two rules as described below:

```
if Data(Head(L)) = true then
S := Push(Head(L), S)
L := Tail(L)
endif
```

```
if Oper(Head(L)) = true  then
if Arg1 = undef  then
Arg1 = Top(S)
S := Pop(s)
elsif Arg2 = undef  then
Arg2 = Top(S)
S := Pop(S)
else S:= Push(Apply(Head(L), Arg1, Arg2), S)
L := Tail(L)
Arg1 := undef
Arg2 := undef
endif
endif
```

Execution (Run)

At the beginning the stack is empty. With the example given, the stack goes through the following states:

() (1) (23 1) (24) (45 24) (6 45 24) (51 24) (1224)

The final result is stored in the stack then the computation finishes.

A.14.2 A TURING MACHINE

Lets consider a generic Turing machine.

Universes (Domains):

The following domains are proposed for the Turing machine:

- **Control:** Distinguished elements are *InitialState* and *CurrentState*.
- **Char:** A distinguished element is *Blank*.
- **Displacement:** +1, -1.
- **Tape:** All tapes of Char. A distinguished element is Head.

Functions:

The following functions can be defined:

- **Move:** Tape x Displacement → Tape

- **NewState:** Control x Char → Control
- **NewChar:** Control x Char → Char
- **NewShift:** Control x Char → Displacement
- **TapeCont::** Tape → Char

Program of the ASM:

The program consists of a rule as described below:

```
[do in-parallel]
  CurrentState := NewState(CurrentState, TapeCont(Head))
  TapeCont(Head):= NewChar(CurrentState, TapeCont(Head))
  Head := Move(Head, Shift(CurrentState, TapeCont(Head))
[enddo]
```

Execution (Run)

At the beginning the *CurrentState* is *InitialState*. All the rules fire every time in a simultaneous fashion.

A.15 SEMANTICS

Formal semantics is concerned with rigorously specifying the meaning, or behaviour, of programs, pieces of hardware. There are two reasons for the need of rigour:

- It can reveal ambiguities and subtle complexities in apparently crystal clear defining documents.
- It can form the basis for implementation, analysis and verification.

It is important to distinguish between the syntax and semantics of languages:

- **Syntax:** is concerned with the grammatical structure of a language. The arrangement of the words in the sentences.
- **Semantics:** is concerned with the meaning of grammatically correct programs.

There are three main different approaches for the specification of semantics:

- **Operational Semantics:** The meaning of a construct is specified the computation it induces when it is executed on a machine. In particular, it is of interest how the effect of a computation is produced. There are two types of operational semantics:

- **Natural Semantics:** its purpose is to describe how the overall results of executions are obtained.
- **Structural Operational Semantics:** its purpose is to describe how the individual steps of the computation take place.
- **Denotational Semantics:** Meanings are modeled by mathematical objects that represent the effect of executing the constructs. Thus, only the effect is of interes, not how is obtained.
- **Axiomatic Semantics:** Specific properties of the effect of executing the constructs are expressed as assertions. Thus, there may be aspects of the executions that are ignored.

EPILOGUE:

ADVICE, LIKE YOUTH, PROBABLY JUST WASTED ON THE YOUNG

Inside every adult lurks a graduation speaker dying to get out, some world-weary pundit eager to pontificate on life to young people who'd rather be Rollerblading. Most of us, alas, will never be invited to sow our words of wisdom among an audience of caps and gowns, but there's no reason we can't entertain ourselves by composing a Guide to Life for Graduates.

I encourage anyone over 26 to try this and thank you for indulging my attempt.

“Ladies and gentlemen of the class of '97.

Wear Sunscreen.

If I could offer you only one tip for the future, sunscreen would be it. The long-term benefits of sunscreen have been proved by scientists, whereas the rest of my advice has no basis more reliable than my own meandering experience. I will dispense this advice now.

Enjoy the power and beauty of your youth. Oh, never mind. You will not understand the power and beauty of your youth until they've faded. But trust me, in 20 years, you'll look back at photos of yourself and recall in a way you can't grasp now how much possibility lay before you and how fabulous you really looked. You are not as fat as you imagine.

Don't worry about the future. Or worry, but know that worrying is as effective as trying to solve an algebra equation by chewing bubble gum. The real troubles in your life are apt to be things that never crossed your worried mind, the kind that blindside you at 4 pm on some idle Tuesday.

Do one thing every day that scares you.

Sing.

Don't be reckless with other people's hearts. Don't put up with people who are reckless with yours.

Floss.

Don't waste your time on jealousy. Sometimes you're ahead, sometimes you're behind. The race is long and, in the end, it's only with yourself. Remember compliments you receive. Forget the insults. If you succeed in doing this, tell me how. Keep your old love letters. Throw away your old bank statements.

Stretch.

Don't feel guilty if you don't know what you want to do with your life. The most interesting people I know didn't know at 22 what they wanted to do with their lives. Some of the most interesting 40-year- olds I know still don't.

Get plenty of calcium. Be kind to your knees. You'll miss them when they're gone.

Maybe you'll marry, maybe you won't. Maybe you'll have children, maybe you won't. Maybe you'll divorce at 40, maybe you'll dance the funky chicken on your 75th wedding anniversary. Whatever you do, don't congratulate yourself too much, or berate yourself either. Your choices are half chance. So are everybody else's.

Enjoy your body. Use it every way you can. Don't' be afraid of it or of what other people think of it. It's the greatest instrument you'll ever own.

Dance, even if you have nowhere to do it but your living room.

Read the directions, even if you don't follow them.

Do not read beauty magazines. They will only make you feel ugly.

Get to know your parents. You never know when they'll be gone for good. Be nice to your siblings. They're your best link to your past and the people most likely to stick with you in the future.

Understand that friends come and go, but with a precious few you should hold on. Work hard to bridge the gaps in geography and lifestyle, because the older you get, the more you need the people who knew you when you were young.

Live in New York City once, but leave before it makes you hard. Live in Northern California once, but leave before it makes you soft.

Travel.

Accept certain inalienable truths. Prices will rise. Politicians will philander. You, too will get old. And when you do, you'll fantasize that when you were young, prices were reasonable, politicians were noble, and children respected their elders.

Respect your elders.

Don't expect anyone else to support you. Maybe you have a trust fund. Maybe you'll have a wealthy spouse. But you never know when either one might run out.

Don't mess too much with your hair or by the time you're 40 it will look 85. Be careful whose advice you buy, but be patient with those who supply it. Advice is a form of nostalgia. Dispensing it is a way of fishing the past from the disposal, wiping it off, painting over the ugly parts and recycling it for more than it's worth.

But trust me on the sunscreen."

--Mary Schimch